Compositional Concurrent Program Verification with RGITL

Dissertation
Zur Erlangung des Doktorgrades Dr. rer. nat.
Institut für Software & Systems Engineering
Fakultät für Angewandte Informatik
Universität Augsburg

Bogdan Tofan

Reviewers:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Alexander Knapp
Day of Defense:	July 11, 2014

Abstract

This thesis takes up the challenge of formal specification and verification of concurrent programs with shared memory. To this end, we use the logic Rely-Guarantee Interval Temporal Logic (RGITL) that has been natively implemented in the interactive theorem prover KIV. The logic incorporates several techniques that make the verification of such concurrent programs more tractable. In particular, temporal logic makes it possible to intuitively specify both safety and liveness requirements for these programs. Moreover, compositional reasoning breaks down the verification of a property of a concurrent program into smaller proof obligations that must be shown for the constituent subprograms.

We mechanically derive various rely-guarantee rules for the compositional verification of partial/total correctness and absence of deadlock of concurrent programs in RGITL. Moreover, we evaluate the practical use of the logic in the domain of highly concurrent data structure implementations: We derive novel compositional proof methods for the central safety/liveness conditions of linearizability and lock-freedom. Finally, we show the application of our methods to verify a number of intricate data structure implementations correct.

Acknowledgment

This thesis has evolved over a period of over 4 years in which I have worked in the formal methods group of the Institute for Software and Systems Engineering at Augsburg University. First of all, I thank Prof. Dr. Wolfgang Reif for his support to work on this interesting topic. I am also grateful to Dr. Gerhard Schellhorn who has constantly and generously spent his time to discuss the technical problems that I came across. His frank and insightful comments have been a steady influence which has led this work into right directions.

I thank Dr. Michael Balser and my former colleague Dr. Simon Bäumler who have paved the way for this thesis with their own work. Furthermore, I acknowledge Jörg Pfähler and Stefan Schödel for their work with RGITL which has led to several improvements. Thanks also to Rico Amslinger for his work on testing concurrent data structure implementations that has broadened our view of strict formal verification.

I am grateful to Dr. Kurt Stenzel, Dr. Dominik Haneberg and Gidon Ernst for their support with my teaching activities. Thanks to all the rest of my colleagues for their helpfulness and the good working atmosphere: Rositta Bingger, Alexandra Dreher, Doris Graziani, Nina Moebius, Hella Seebach, Gerrit Anders, Andreas Angerer, Marian Borek, Benedikt Eberhardinger, Axel Habermaier, Alwin Hoffmann, Kuzman Katkalov, Johannes Leupolz, Ludwig Nägele, Florian Nafz, Andreas Schierl, Florian Siefert, Peter Fischer, Alexander Schiendorfer, Jan-Philipp Steghöfer and Michael Vistein.

Many thanks to Prof. Dr. Alexander Knapp for various valuable remarks on our work. Thanks to Prof. Dr. Cliff Jones for his comments on our approach and his valuable hints to related work. Thanks to Prof. Dr. Ben Moszkowski for his permanent strive to foster the ideas of interval temporal logic, some of which are part of our logic, and his thorough reading of some of our papers. Many thanks to Prof. Dr. Lindsay Groves for his comments and discussion on our work on proving linearizability. Thanks to Dr. Brijesh Dongol for interesting discussions on non-blocking progress conditions. I also thank Oleg Travkin for his work on proving linearizability with KIV. Thanks to Prof. Dr. John Derrick and Prof. Dr. Heike Wehrheim for interesting discussions on various notions of correctness for concurrent data structures. Thanks to the anonymous reviewer of [94] that helped us to better understand the importance of unfair scheduling for lock-freedom; I also appreciate the constructive remarks of the reviewers of [7, 20, 89, 90, 94–96, 99] that have helped to improve our work.

Last and most importantly, I value the loving understanding of my wife Camelia and our two children Lara and Luca.

To Lara and Luca.

Outline

The design of concurrent programs is a delicate issue since it must cope with a variety of possible interactions between running threads. For similar reasons, the formal specification and verification of concurrent programs is a challenging task. The benefit of formal analysis is a more precise understanding and an increased confidence in the correctness of these implementations that can be very subtle. This is particularly important as concurrent programs are part of many critical computer systems.

Several approaches have been developed throughout the last decades to tackle the difficulties of concurrent program verification, ranging from early general approaches without direct tool support [53, 78], to recent fully automatic specialized techniques [104,107]. We have taken a general, mechanized approach based on interactive theorem proving that gives a human proof engineer the freedom to guide the verification tool whenever necessary. This helps to overcome some of the drawbacks of purely manual and fully automatic approaches, which are either hard to use for non-expert users or restricted to automatically solving only specific problems.

Our specification and verification approach is based on the concurrent temporal logic RGITL that can express a wide range of safety ("nothing bad happens throughout program execution") and liveness properties ("eventually a program does something good"). The logic integrates several ideas from interval temporal logic (ITL) [74], such as having programs as formulas with compositional operators for sequential programs. It extends ITL with explicit blocking, compositional interleaving and recursive procedures. The calculus of RGITL is based on symbolic execution of explicit programs and temporal formulas with "a little induction"; it has been natively implemented [5] in the interactive verifier KIV [62]. To ensure the soundness of RGITL, we have mechanically specified and verified the semantic foundation of the logic in higher-order logic (HOL) [57].

One central feature of our approach is compositionality. We exploit this feature here to derive various rely-guarantee (RG) rules [53, 109] which allow us to examine general correctness properties of concurrent programs by looking at their sequential subprograms only. We describe an embedding of RG reasoning into RGITL for proving partial/total correctness and absence of deadlock. Moreover, we derive state-local versions of such rules that reduce the overall program state with an arbitrary finite number of local states to a fixed small number of local states only. This is particularly useful for concurrent data type implementations where components have similar behaviors. We provide mechanized soundness proofs for all of these rules [59].

The main application domain of the introduced theory are highly concurrent data structure implementations which either use fine-grained locking mechanisms or no locks at all to ensure a high degree of parallelism. The shift from mono-core processors to the

nowadays prevalent multi-core processors has led to an increased significance of such implementations as they can better benefit from these architectures than algorithms with coarse-grained locking. Consequently, fine-grained algorithms have become part of widely used concurrency libraries of modern programming languages such as Threading Building Blocks for C++, System.Collections.Concurrent for C#, java.util.concurrent for Java, or scala.collection.concurrent for Scala.

The practical use of RGITL in the domain of highly concurrent data structures has led to the following central results of this work: On the one hand, we have derived novel proof methods for the central safety/liveness properties of linearizability [48] and lock-freedom [67] in the logic. In particular, we derive a generic proof method for linearizability that combines the complete proof method of possibilities [48] with RG reasoning, and a refinement-based proof method for an important subclass of linearizable algorithms. Furthermore, we derive compositional proof obligations for lock-freedom that apply to a wide range of lock-free algorithms.

On the other hand, we have evaluated the practical use of our methods by verifying intricate concrete algorithms correct, most of which had no (mechanized) formal verification before: For instance, our highly concurrent multiset algorithm with intricate linearization points (Section 9.4), or the lock-free stack that uses the memory reclamation technique of hazard pointers [68] (Section 12.2). Moreover, we show how additional techniques such as ownership annotations [76] – known from the verification of object-oriented programs – and separation logic [85] can be used in RGITL to reduce the specification/proof effort in some cases.

As usual in computer science nowadays, several results of this thesis have already been published in international conferences and journals [7, 20, 89, 90, 94–96, 98, 99]. Of course, this thesis is based on our descriptions there: Each chapter typically references our respective paper(s) in the introduction and provides new/further explanations where appropriate.

The rest of this work is structured in three parts: Part I introduces our logic RGITL which forms the logical fundament for the remaining two parts. Part II describes how RG reasoning is done in the logic and Part III derives and applies our proof methods for linearizability and lock-freedom which are all based on RG reasoning. A comparison of this work with our own previous work as well as other related approaches is given at the end of each part. Since all specifications and proofs that we describe here have been mechanized in KIV, we take a semi-formal approach in the presentation, only sketching the essential parts of our proofs for the sake of brevity and readability. Interested readers are, however, referred to our online descriptions of the mechanized specifications/proofs for full details throughout the thesis.

Contents

Part I.

The Logic RGITL

The first part of this work introduces the logic Rely-Guarantee Interval Temporal Logic (RGITL). The design of the logic pursues two central goals: i) The logic offers a framework in which we can specify and verify compositional proof methods (rules) for concurrent programs such as rely-guarantee rules or compositional proof methods for (domain) specific properties such as linearizability or lock-freedom. We exploit this feature of the logic in parts two and three of this work. ii) It makes it possible to specify and verify concrete concurrent programs correct, either by directly executing them or by first applying a specific proof method that has been specified in the logic.

In previous work [5], RGITL has been natively implemented in the interactive verifier KIV [62] and the basic rules of the logic have been verified correct on a semantic level on paper. In addition, we have now mechanically checked these proofs in higher-order-logic using KIV as we describe online at [57]. This effort has led to several minor adaptations of the logic and its implementation.

In the following, we try to give a comprehensible exposition of the logic. The structure of the rest of Part I is as follows: Chapter 1 introduces the syntax and semantics of RGITL. In particular, we introduce the fundamental compositionality rules of the logic and prove their correctness. These rules are essential to achieve goal i) above. Then Chapter 2 defines the calculus rules for symbolic execution of sequential and concurrent programs with induction and illustrates them using simple examples, according to goal ii) above. Finally, we conclude in Chapter 3 with a discussion of related and possible future work.

1. Syntax and Semantics

The semantics of RGITL is based on intervals which are essentially finite or infinite sequences of variable evaluation functions. Different from intervals in ITL [74], our interval semantics has built in arbitrary environment transitions, similar to reactive sequences in [86]. This is the key for the compositional reasoning about a concurrent program by looking at its constituent subprograms only and reusing these individual proofs in the specific context of the other programs and the overall environment. This chapter widely corresponds to our recent articles [89, 90].

The structure of this chapter is as follows: Section 1.1 informally motivates the basic principles of RGITL – compositional reasoning and symbolic execution with induction – using a standard example. Section 1.2 then formally defines the syntax of the logic and Section 1.3 its semantics. In particular, we give generic proof rules for compositional reasoning and verify them correct w.r.t. the introduced semantics. Section 1.4 focuses on the compositional semantics of interleaving and illustrates it with simple examples. Finally, Section 1.5 introduces the abstract programming language of RGITL and briefly discusses the KIV approach to structure specifications and to generate proof obligations.

1.1. Introduction

1.1.1. A Simple Concurrent Search Algorithm

Figure 1.1 shows a simple concurrent algorithm [79] in pseudo code. It calculates the minimal index in an array of elements such that a given predicate *pred* evaluates to true. The central idea behind the algorithm is:

1. Divide the overall search task in two distinct search tasks that can be executed by two processes in parallel.

2. Merge the two individual results to yield the overall result by computing their minimum.

The first process `EVEN` searches the even indices and the second process `ODD` the odd indices of the shared array *Ar*, respectively. If one of the two processes finds a position where *pred* holds, it terminates after notifying the other process by setting the shared variable *OutE*/*OutO* to its current position. The other process then only continues its search if its current position is less than the found index of the other process.

For the parallel FIND algorithm we are interested in compositional proofs (see below) of the following global correctness properties.

```
sort elem;
var Ar: array of elem;
var pred: elem -> bool;
var OutE, OutO := # Ar;

FIND()
  EVEN(OutE) || ODD(OutO);
  return min(OutE, OutO)

EVEN(OutE)
  var i := 0;
  while i < OutO {
    if (pred(Ar[i])) {
      OutE := i;
      return
    } else
      i := i + 2
  }

ODD(OutO)
  var i := 1;
  while i < OutE {
    if (pred(Ar[i])) {
      OutO := i;
      return
    } else
      i := i + 2
  }
```

Figure 1.1.: A Concurrent Search Algorithm

- Safety:
  - The FIND algorithm never accesses the array out of bounds.
  - Whenever FIND terminates, it either returns the minimal index where *pred* holds, or the length of the array if *pred* does not hold at any array position.
  - The program never runs into a deadlock (which is trivial here, since it does not use any locks).
- Liveness: Each execution of FIND terminates, i.e., FIND never runs into a livelock.

1.1.2. Compositional Reasoning

For program verification we want to incorporate the "divide and conquer" principle that underlies the parallel FIND algorithm, as well as many other concurrent algorithms. That is, the compositional proof of a global property for a concurrent program, first decomposes the overall property into smaller proof obligations for its constituent subprograms, which can then be composed to yield the original property again. Our experience shows that the effort to decompose the proof of a global property into smaller pieces is worthwhile, since noncompositional proofs that directly tackle global properties are typically not practicable even for small concurrent programs.

Before we give the central compositionality rule of RGITL, rule (1.1) below, we introduce some standard notation: RGITL uses the well-known sequent calculus as

the basic assertion language, which writes assertions (proof goals) as *sequents*

$$\Gamma \vdash \Delta$$

The *antecedent* Γ and the *succedent* Δ of a sequent are two (possibly empty) lists of formulas. A sequent $\Gamma \vdash \Delta$ is a convenient notation for the formula

$$(\bigwedge \Gamma) \rightarrow (\bigvee \Delta)$$

It is valid if the conjunction of all formulas from the antecedent implies the disjunction of all formulas from the succedent. An empty antecedent corresponds to formula *true* and an empty succedent to *false*, respectively. Sequents are implicitly universally closed.

Rules in sequent calculus have the form

$$\frac{p_1 \dots p_n}{c}$$

where sequent c is the conclusion, and sequents $p_1 \dots p_n$ are the premises. A rule with no premises is an axiom; a rule is sound, if valid premises above the line imply a valid conclusion. Hence, axioms are valid per se. Rules are typically applied bottom-up, reducing a proof goal c to (simpler) proof goals p_1 to p_n.

In RGITL programs are formulas, a concept that is adopted from ITL [74]. Proof obligations for programs are thus simply written as sequents such as

$$\alpha \vdash \varphi$$

which requires that program α satisfies property φ.

With these preliminaries, we can now define the following central proof rule of RGITL, which allows for compositional reasoning about sequential and parallel programs.

$$\frac{\alpha_1 \vdash \varphi_1 \qquad \alpha_2 \vdash \varphi_2 \qquad \varphi_1 \odot \varphi_2 \vdash \psi}{\alpha_1 \odot \alpha_2 \vdash \psi} \tag{1.1}$$

The rule states that to prove a property ψ for a composed program $\alpha_1 \odot \alpha_2$ requires to prove suitable subproperties φ_1 and φ_2 for the subprograms α_1 and α_2 (premises 1 and 2), plus to show that the composition $\varphi_1 \odot \varphi_2$ of these *subproperties* satisfies the overall property ψ (premise 3). The rule is sound for sequential and parallel composition operators $\odot$ and arbitrary programs α and formulas φ, ψ.[1]

The compositionality rule (1.1) is the central rule that makes it possible to derive compositional proof methods (rules) for global correctness properties of concurrent programs in the logic. Thereby the premises 1 and 2 for the subcomponents of the program become proof obligations of the proof method and premise 3 is generically derived (once for all) in the soundness proof of the rule.

[1] More general versions that replace the program α with an arbitrary temporal formula also hold and are given in Sections 1.3 and 1.4.

Example 1.1 (Sketch of Compositional Proof with Rule (1.1))
We now roughly sketch a compositional proof for an arbitrary global property ψ_{FIND} of the FIND algorithm: First, the compositionality rule (1.1) is applied to the sequential composition operator ";" which gives three new premises (1) - (3):

$$\frac{\begin{array}{ll}(\texttt{EVEN} \parallel \texttt{ODD}) \vdash \varphi_{(\texttt{EVEN} \parallel \texttt{ODD})} & (1)\\ \mathit{Out} := \mathit{min}(\mathit{OutE}, \mathit{OutO}) \vdash \varphi_{:=} & (2)\\ \varphi_{(\texttt{EVEN} \parallel \texttt{ODD})};\ \varphi_{:=} \vdash \psi_{FIND} & (3)\end{array}}{\mathit{FIND} \vdash \psi_{FIND}}\ \textit{with (1.1) for } \odot = \ ;$$

Premise (1) requires to show that the interleaved subprogram (EVEN ∥ ODD) *satisfies a suitable subproperty $\varphi_{(\texttt{EVEN} \parallel \texttt{ODD})}$. The proof of premise (1) applies the compositionality rule (1.1) again (see below). Premise (2) introduces a suitable subproperty $\varphi_{:=}$ for the assignment $\mathit{Out} := \mathit{min}(\mathit{OutE}, \mathit{OutO})$ such that the sequential composition of $\varphi_{(\texttt{EVEN} \parallel \texttt{ODD})}$ and $\varphi_{:=}$ yields the overall property ψ_{FIND} according to premise (3).*

We further verify premises (2) and (3) using symbolic execution for sequential programs, as we explain below. For the proof of premise (1) we must again apply rule (1.1) which gives three new proof obligations (4) - (6):

$$\frac{\begin{array}{ll}\texttt{EVEN} \vdash \varphi_{\texttt{EVEN}} & (4)\\ \texttt{ODD} \vdash \varphi_{\texttt{ODD}} & (5)\\ (\varphi_{\texttt{EVEN}} \parallel \varphi_{\texttt{ODD}}) \vdash \varphi_{(\texttt{EVEN} \parallel \texttt{ODD})} & (6)\end{array}}{(\texttt{EVEN} \parallel \texttt{ODD}) \vdash \varphi_{(\texttt{EVEN} \parallel \texttt{ODD})}}\ \textit{with (1.1) for } \odot = \ \parallel$$

Premises (4) and (5) define subproperties $\varphi_{\texttt{EVEN}}$ and $\varphi_{\texttt{ODD}}$ that are shown for the sequential programs EVEN *and* ODD *using symbolic execution. Finally, premise (6) requires to show that the interleaving of these subproperties implies the overall property $\varphi_{(\texttt{EVEN} \parallel \texttt{ODD})}$. Subproperties $\varphi_{\texttt{EVEN}}$ and $\varphi_{\texttt{ODD}}$ can be generically defined to be compositional as rely-guarantee assertions (see Section 4.2), i.e., we do not have to prove premise (6) for a specific case study. Instead, it follows directly from the generic correctness argument of a corresponding rely-guarantee rule (see Part II).*

In summary, applying rule (1.1) *(plus a suitable rely-guarantee rule) essentially reduces the verification of ψ_{FIND} for the interleaved program FIND to verifying individual subproperties of its constituent sequential programs* EVEN *and* ODD*. The verification of premises (1) - (6), including the correctness proofs for rely-guarantee rules, is entirely carried out in RGITL.*

1.1.3. Symbolic Execution with Induction

RGITL supports the verification of proof obligations, such as premises (4) and (5) from the example above, by stepping forwards through the program code and the (temporal) formulas of a sequent. That is, we adapt the principles of symbolic execution for sequential programs – calculating the strongest post-condition for a given program

statement and a pre-condition [12, 36, 55] – to a concurrent setting with temporal logic.[2]

As an example, consider a simple assignment $x := t; \gamma$ that sets a variable x to the current value of a term t, with some rest program γ, a pre-condition formula *Pre* and a post-condition *Post*. A symbolic execution step of this program works according to the following rule:

$$\frac{Pre_x^{x_0} \wedge x = t_x^{x_0},\ \gamma\ \vdash Post}{Pre,\ (x := t; \gamma)\ \vdash Post}\ SymbExec$$

That is, a symbolic execution step computes the strongest post-condition $Pre_x^{x_0} \wedge x = t_x^{x_0}$ of the assignment $x := t$ and pre-condition *Pre* and discards the first program statement. According to the resulting premise of the rule above, this strongest post-condition becomes the new pre-condition for the rest program. The new variable x_0 represents the value of x before the assignment. It is used to syntactically replace each (free) occurrence of x in *Pre* and t, denoted as $Pre_x^{x_0}$ and $t_x^{x_0}$, respectively.[3]

Example 1.2 (Symbolic Execution of Sequential Programs)
For the simple assignment $i := i + 2$, the pre-condition formula $i = 0$ and the post-condition $i = 2$, a symbolic execution step gives

$$\dfrac{\dfrac{\dfrac{}{i = 2 \vdash i = 2}\ Axiom}{i_0 = 0 \wedge i = i_0 + 2 \vdash i = 2}\ Arithmetic}{i = 0,\ i := i + 2 \vdash i = 2}\ SymbExec$$

where the resulting premise can be easily reduced to an axiom of sequent calculus with basic arithmetic.

In Example 1.2, the pre-condition specifies exactly one initial state and symbolic execution directly corresponds to the interpretation of a program with a debugger, i.e., just testing the correctness of a program for a specific input. Weakening the pre-condition to specify several initial states (e.g., using the formula $i \geq 0$) makes a smooth transfer from simple program testing to full program verification with arbitrary variable domains. For instance, it is easy to prove that incrementing i by 2 results in a value $i \geq 2$ if i was non-negative initially:

$$\dfrac{\dfrac{\dfrac{}{i \geq 2 \vdash i \geq 2}\ Axiom}{i_0 \geq 0 \wedge i = i_0 + 2 \vdash i \geq 2}\ Arithmetic}{i \geq 0,\ i := i + 2 \vdash i \geq 2}\ SymbExec$$

In a concurrent setting with temporal logic, symbolic execution must additionally take environment behavior into account. A rough idea how to transfer symbolic execution to a concurrent setting with temporal logic is as follows: Computing the strongest

[2] Backward reasoning would calculate backwards the weakest pre-condition for a program statement and a given post-condition *Post* [50].

[3] For an empty rest program, the rule simply has the premise $Pre_x^{x_0} \wedge x = t_x^{x_0} \rightarrow Post$.

post-condition of the first step of a program α reduces an initial proof obligation

$$Pre, \alpha, E \vdash \varphi$$

where E is a formula that describes the behavior of α's environment and φ is an arbitrary (temporal) formula to be verified, to a new proof obligation

$$Pre_1, \alpha_1, E_1 \vdash \varphi_1$$

such that

- The new program α_1 is α without its first step.
- Formula Pre_1 is the new pre-condition of α_1. It is an assertion for the state after the first step of α *and* possibly further steps of α's environment. That is, different from the sequential setting, the strongest post-condition Pre_1 now additionally considers the behavior of α's environment according to E.
- Formula E_1 is the new environment assumption that follows from E for the rest of the program. (This is often E again).
- Formula φ_1 is the property that must be proved for the remaining program α_1.

Example 1.3 (Symbolic Execution with Concurrency)
The symbolic execution of the assignment $i := i + 2$ in an empty environment E that never changes the local variable i, directly corresponds to the sequential case above.

If E permits arbitrary concurrent changes to variable i, i.e., $E \equiv true$, the strongest post-condition for the assignment must be weakened and a new variable i_1 must be introduced to store the state right after the assignment, but before the environment executes. This gives

$$\frac{i_0 \geq 0 \wedge i_1 = i_0 + 2,\ true \vdash i \geq 2}{i \geq 0,\ i := i + 2,\ true \vdash (i \geq 2)_{Post}}\ \textit{SymbExec in Arbitrary Environment}$$

and the post-condition $i \geq 2$ can no longer be shown, since the environment might decrement the value of i (i.e., i_1) such that it becomes less than 2. Only if E would ensure that i is never decremented (hence $i_1 \leq i$), we could verify the premise above correct. These rough ideas will be formalized in Theorem 2.2.

Induction. Symbolic execution works for a wide class of (concurrent) programs and temporal properties. However, it is not sufficient: To deal with (possibly infinite) loops, further rules for induction must be added to the calculus. For safety formulas φ, we can give general induction rules which basically reduce the verification of φ for an infinite program execution, to the verification of φ for a finite execution only. For liveness formulas, however, a specific term for induction (a variant) must be provided by the proof engineer. For instance, for each subprogram of the FIND algorithm from Figure 1.1 we can use the variant $\#Ar - i$ that decreases with every loop iteration to prove termination. Of course, environment behavior must also be taken into account in these termination proofs. Otherwise, if an individual search operation executes in an environment that can increase the size of the array, then it will possibly not terminate.

$$
\begin{aligned}
e ::= {} & v \mid x' \mid x'' \mid op \mid e(\underline{e}) \mid e_1 = e_2 \mid \lambda\, \underline{u}.\, e \mid \forall\, \underline{v}.\, \varphi \mid \\
& \varphi_1 \textbf{ until } \varphi_2 \mid \varphi_1; \varphi_2 \mid \varphi^* \mid \textbf{A}\ \varphi \mid \textbf{last} \mid \circ\, \varphi \mid \textbf{blocked} \mid \\
& \varphi_1 \parallel \varphi_2 \mid \varphi_1 \parallel_{\mathsf{nf}} \varphi_2 \mid \texttt{PROC}(\underline{e}; \underline{x})
\end{aligned}
$$

Figure 1.2.: Syntax of Expressions in RGITL

1.2. Syntax

As usual, we start the exposition of our logic RGITL, by defining the admissible syntax of expressions. It is based on a higher-order logic signature SIG which consists of the three finite sets S, OP and $Proc$.

- S is the set of sorts that includes the boolean sort *bool* by default. This set is used to construct the set of types T which is the set of all sorts and function types $\underline{t} \to t$, where $t \in T$ and $\underline{t} = t_1, \ldots, t_n$ such that t_1 to t_n are in T. As usual in higher-order logic, functions can return functions and may also use them as parameters.

- OP is the set of typed operators $op : t$ which includes the standard boolean operators such as $false, true : bool$, $\neg\ : bool \to bool$ and $\vee : bool, bool \to bool$.

- $Proc$ is the set of typed procedure names $\texttt{PROC}\!:\!\underline{t_1};\underline{t_2}$, where PROC is the procedure name, $\underline{t_1}$ indicates the types of the input (or call by value) parameters and $\underline{t_2}$ denotes the types of the in-output (or call by reference) parameters of PROC.

Expressions can use flexible (also called "dynamic") variables x, y, z from the set of flexible variables F and static variables u from the set of static variables St. Flexible variables can change their value during the execution of a program, while static variables remain unchanged. Static variables are often used to capture the "historic" value of a flexible variable that evolves during symbolic execution. In concrete formulas we typically follow the ITL convention to write flexible variables with a capital letter and static variables lowercase. Variables that are either flexible or static are written v.

Figure 1.2 defines the syntax grammar of higher-order and temporal logic expressions e over a signature SIG and the sets of variables F and St. Expressions must meet the following constraints: The usual typing constraints must be satisfied, e.g., in the function application $e(\underline{e})$ the type of e must be a function type with argument types equal to the types of the arguments $\underline{e}$. Moreover, the parameters of lambda expressions (which are "anonymous functions") and quantifiers must all be different variables. Lambda expressions use static variables only, while quantifiers can use both static and flexible variables; the in-output parameters of procedure calls are pair-wise distinct flexible variables. As usual, a formula is an expression of boolean type which we denote as φ.

We give the intuitive semantics of expressions in the following: Flexible variables x can be primed x' and double primed x''. The result of priming is not a new variable, but simply an expression that refers to x in a future state (see below). Similar to the chop and chop-star operators from ITL, the sequential composition operator $\varphi_1; \varphi_2$ combines two formulas sequentially, and the star operator φ^* finitely (possibly zero times) or infinitely often iterates a formula φ. Universal path quantification is denoted as $\mathbf{A}\,\varphi$. Formula **last** characterizes termination. The strong next operator $\circ\,\varphi$ states that formula φ will be true in the next execution state. Formula **blocked** indicates a blocked program step that occurs when a program waits for a lock that is taken by some other process. Formulas $\varphi_1 \parallel \varphi_2$ and $\varphi_1 \parallel_{\mathsf{nf}} \varphi_2$ denote weak-fair/unfair interleaving of φ_1 and φ_2. Procedure calls have the form $\texttt{PROC}(\underline{e}; \underline{x})$ such that $\texttt{PROC}\colon \underline{t_1}; \underline{t_2}$ is in $Proc$ and the input parameters $\underline{e}$ have types $\underline{t_1}$ and the in-output parameters $\underline{x}$ are of types $\underline{t_2}$. Occasionally, we use $\texttt{PROC}(\underline{x})$ as a short form for $\texttt{PROC}(;\underline{x})$ when no input parameters are required.

The free variables $\mathrm{free}(e)$ of an expression e are defined in the standard way. The free variables of primed and double primed variables are

$$\mathrm{free}(x') = \mathrm{free}(x'') = \{x\}$$

and renaming x to y in x'/x'' gives y'/y''.

In general expressions, the constructs from the grammar in Figure 1.2 can be freely mixed, i.e., temporal operators may appear in equalities and so forth. Sometimes, we must restrict ourselves to specific types of expressions. The following definition introduces special types of expressions that will be used in future sections:

Definition 1.1 (Special Types of Expressions)

- *A higher-order expression is an expression that only uses the constructs from the first line from Figure 1.2.*
- *A state expression is a higher-order expression without primed and double primed variables.*
- *A state formula is a state expression of type bool.*
- *A static expression is a higher-order expression without flexible variables (neither free nor bound).*
- *A static formula is a static expression of type bool.*

Finally, we use the following binding conventions: Propositional connectives have the following ascending binding priority

$$\leftrightarrow \;\prec\; \rightarrow \;\prec\; \vee \;\prec\; \wedge \;\prec\; \neg$$

and bind weaker than all temporal operators (**until**, ;, *, $\mathbf{A}$, $\circ$, $\parallel$, $\parallel_{\mathsf{nf}}$). Quantifiers bind as far to the right as possible.

1.3. Semantics

Now to the semantics of the previously introduced syntax which maps syntactic objects to mathematical (algebraic) objects: The semantics of a signature $SIG = (S, OP, Proc)$ is a SIG-algebra $\mathcal{A}$ which defines the semantics of every sort s from S as a non-empty carrier set A_s. The set A_{bool} is {ff, tt} with semantic false and true values, respectively. The semantics of a function type is the set of all functions of that type. Furthermore, $\mathcal{A}$ interprets each operator symbol $op{:}t$ from the set OP as a total function $op^{\mathcal{A}} \in A_t$. The predefined boolean operators have standard semantics, e.g., $false^{\mathcal{A}} = \text{ff}$ and $true^{\mathcal{A}} = \text{tt}$. A semantics for procedure calls $\texttt{PROC} \in Proc$ is defined below.

All expressions e from the syntax grammar in Figure 1.2 are interpreted relative to a SIG-algebra $\mathcal{A}$ that we leave implicit in the following and an *interval* I, denoted as $[\![e]\!](I)$. An interval is a finite or infinite sequence of the form

$$I = (I(0), I(0)_b, I'(0), I(1), I(1)_b, I'(1), I(2), \ldots)$$

where every $I(k)$ and $I'(k)$ is a state function (or simply a state) that maps variables to values. The state transition from $I(k)$ to $I'(k)$ is called a program (or system) transition, whereas the transition $I'(k)$ to $I(k+1)$ from a primed to the subsequent unprimed state is an environment transition. Thus intervals alternate between program and environment transitions, similar to reactive sequences in [86]. Finite intervals have an odd number of states. The boolean flag $I(k)_b$ denotes whether the program transition from $I(k)$ to $I'(k)$ is *blocked*. For a blocked transition, the flag $I(k)_b$ is tt and $I(k)$ is equal to $I'(k)$, i.e., the transition stutters. Static variables do not change in any (program or environment) transition of an interval.

The semantics $[\![e]\!](I)$ of an expression e of type t relative to an interval I is an element of A_t, e.g., the semantics $[\![\varphi]\!](I)$ of a formula φ is either ff or tt. Before we actually define the semantics of the expressions, we define some auxiliary notions for an interval I:

Definition 1.2 (Simple Interval Notions & Functions)

- *I is empty if it has just one (initial) state $I(0)$.*
- *A formula φ holds over I (or I satisfies φ) if its semantics $[\![\varphi]\!](I)$ is* tt*, denoted as $I \models \varphi$.*
- *A formula φ is valid if it holds over all intervals, written $\models \varphi$.*
- *The length of I is written $|I|$: If I has $2n+1$ states then its length is n. (Hence, empty intervals with just one initial state have length 0.) If I is infinite, its length is ∞.*
- *For $m \leq n \leq |I|$, $I_{[m..n]}$ denotes the subinterval from state $I(m)$ to $I(n)$ inclusive; for an empty interval, we define $I_{[0..n]} := I$.*
- *For $m \leq |I|$, $I_{[m..]}$ denotes the postfix of I from state $I(m)$.*

- *Concatenation of I with interval I_0 is written $I + I_0$ if I is finite and its last state is equal to I_0's initial state.*
- *For a vector of (unprimed flexible or static) variables $\underline{v}$, a value sequence $\sigma = (\sigma(0), \sigma'(0), \sigma(1), \sigma'(1), \ldots)$ consists of tuples of values $\sigma(i)$ and $\sigma'(i)$ of appropriate types. For a static variable v_k, all values $\sigma(i)_k$ and $\sigma'(i)_k$ must be identical.*
- *The value sequence for $\underline{x}$ in I is written $I(\underline{x})$.*
- *The modified interval that maps $\underline{v}$ in each state to the corresponding values in σ is written $I[\underline{v} \leftarrow \sigma]$ for $|\sigma| = |I|$. (Tacitly, we assume that $\sigma(k) = \sigma'(k)$ whenever $I(k)_b$ holds.) Similarly, $I[\underline{u} \leftarrow \underline{a}]$ denotes the modified interval where static variables $\underline{u}$ are replaced with the tuple of values $\underline{a}$.*

Based on Definition 1.2, Figure 1.3 now gives the semantics of expressions, except interleaving that we introduce in Section 1.4. In particular, unprimed flexible variables and static variables v are evaluated over the first state $I(0)$ of an interval. Hence, the semantics of state expressions depends on the first state $I(0)$ only. Primed and double primed variables x' and x'' are evaluated over $I'(0)$ and $I(1)$ respectively, if I is not empty. For an empty interval, both x' and x'' are evaluated over $I(0)$ by convention. The semantics of a function application $[\![e(\underline{e})]\!](I)$ recursively evaluates the semantics of e and its parameters $\underline{e}$ over I. The semantics of a lambda expression $\lambda\ \underline{u}.\ e$ is a function that maps values $\underline{a}$ for the static variables $\underline{u}$ to $[\![e]\!]$ over the modified interval $I[\underline{u} \leftarrow \underline{a}]$. Two expressions are equal if they evaluate to the same semantic value over I. The universally quantified formula $\forall\ \underline{v}.\ \varphi$ evaluates formula φ over the modified interval $I[\underline{v} \leftarrow \sigma]$ for an arbitrary value sequence σ of appropriate length, i.e., the values of v are replaced by arbitrary new values over the entire interval I.

Formula **last** characterizes an empty interval. In a **blocked** state the interval is not empty and the blocked flag $I(0)_b$ holds. The strong next operator $\circ\ \varphi$ holds over nonempty intervals I if φ holds for the postfix $I_{[1..]}$. The until operator φ_1 **until** φ_2 requires formula φ_2 to hold in some postfix $I_{[n..]}$ and φ_1 must hold for each postfix $I_{[m..]}$ with $m < n$. The universal path quantifier $\mathbf{A}\ \varphi$ requires φ to hold for each interval J that starts with $I(0)$. The sequential composition operator $\varphi_1; \varphi_2$ holds if either I is infinite and satisfies φ_1, or there is a finite prefix of I where φ_1 holds and φ_2 holds for the rest of I. The star operator φ^* holds for zero iterations of φ or it splits I into non-empty subintervals $I_{[n_i..n_{i+1}]}$ where φ holds.

Finally, the semantics $\texttt{PROC}^{\mathcal{A}}$ of a procedure $\texttt{PROC} : \underline{t_1}; \underline{t_2}$ is a set of pairs $(\underline{a}, \sigma)$ that describe possible executions of PROC as follows: The procedure starts with a type-consistent vector of initial values $\underline{a}$ for its input parameters and its reference parameters change according to the value sequence σ during execution. Hence, a procedure works on a local copy of its input parameters (changes to the input parameters while the procedure is executing are not visible from the outside) and the procedure can only modify variables from its parameter list.

The compositional semantics of the sequential composition and star operators make it already possible to prove the following central compositionality rules for these two

$$
\begin{aligned}
[\![v]\!](I) &\equiv I(0)(v) \\
[\![x']\!](I) &\equiv I(0)(x) \text{ if } |I| = 0 \text{ and } I'(0)(x) \text{ otherwise} \\
[\![x'']\!](I) &\equiv I(0)(x) \text{ if } |I| = 0 \text{ and } I(1)(x) \text{ otherwise} \\
[\![e(\underline{e})]\!](I) &\equiv [\![e]\!](I)([\![\underline{e}]\!](I)) \\
[\![\lambda\ \underline{u}.\ e]\!](I) &\equiv \underline{a} \mapsto [\![e]\!](I[\underline{u} \leftarrow \underline{a}]) \\
[\![(\varphi \supset e_1; e_2)]\!](I) &\equiv [\![e_1]\!](I) \text{ if } I \models \varphi, \text{ otherwise } [\![e_2]\!](I) \\
I \models e_1 = e_2 &\text{ iff } [\![e_1]\!](I) = [\![e_2]\!](I) \\
I \models \forall\ \underline{v}.\ \varphi &\text{ iff for all } \sigma,\ |\sigma| = |I| : I[\underline{v} \leftarrow \sigma] \models \varphi \\
I \models \textbf{last} &\text{ iff } |I| = 0 \\
I \models \textbf{blocked} &\text{ iff } |I| \neq 0 \text{ and } I(0)_b \\
I \models \circ\ \varphi &\text{ iff } |I| \neq 0 \text{ and } I_{[1..]} \models \varphi \\
I \models \varphi_1\ \textbf{until}\ \varphi_2 &\text{ iff there is } n \leq |I| \text{ with } I_{[n..]} \models \varphi_2 \\
&\quad \text{and for all } m < n\colon I_{[m..]} \models \varphi_1 \\
I \models \mathbf{A}\,\varphi &\text{ iff } \varphi \text{ holds over all intervals } J \text{ with } J(0) = I(0) \\
I \models \varphi_1; \varphi_2 &\text{ iff either } |I| = \infty \text{ and } I \models \varphi_1 \text{ or there is } n \leq |I|\,, n \neq \infty \\
&\quad \text{with } I_{[0..n]} \models \varphi_1 \text{ and } I_{[n..]} \models \varphi_2 \\
I \models \varphi^* &\text{ iff either } |I| = 0 \text{ or there is a sequence } \nu = (n_0, n_1, \ldots), \\
&\quad \text{such that } n_0 = 0 \text{ and for } i + 1 < |\nu| : n_i < n_{i+1} \leq |I| \\
&\quad \text{and } I_{[n_i..n_{i+1}]} \models \varphi. \text{ Whenever } |\nu| < \infty : I_{[n_{|\nu|-1}..]} \models \varphi \\
I \models \texttt{PROC}(\underline{e}; \underline{x}) &\text{ iff } ([\![\underline{e}]\!](I), I(\underline{x})) \in \texttt{PROC}^{\mathcal{A}}
\end{aligned}
$$

Figure 1.3.: Interval Semantics of Expressions (Except Interleaving).

operators (cf. rule (1.1)):

Theorem 1.1 (Compositionality of Sequential Composition)
The following compositionality rule is sound for the sequential composition operator ";":

$$\frac{\varphi_1 \vdash \psi_1 \qquad \varphi_2 \vdash \psi_2 \qquad \psi_1; \psi_2 \vdash \psi}{\varphi_1; \varphi_2 \vdash \psi}$$

Proof The conclusion of the rule is valid if $I \models \varphi_1; \varphi_2$ implies that ψ holds over I, too. If I is infinite and φ_1 holds over I, then the first premise of the rule implies that ψ_1 also holds. By definition of ;, formula $\psi_1; \psi_2$ holds over I and premise 3 ensures $I \models \psi$.

Otherwise, there is a number n such that φ_1 holds for $I_{[0..n]}$ and φ_2 holds for $I_{[n..]}$ by definition of ; . Then premises 1 and 2 ensure ψ_1 / ψ_2 for these subintervals and premise 3 gives $I \models \psi$ using the definition of ; again. This concludes the soundness proof of Theorem 1.1. □

A similar compositionality property also holds for the sequential iteration operator *:

Theorem 1.2 (Compositionality of Sequential Iteration)
The following compositionality rule is sound for the star operator:

$$\frac{\varphi \vdash \psi_1 \qquad \psi_1{}^* \vdash \psi}{\varphi^* \vdash \psi}$$

Proof The conclusion of the rule is valid if for an arbitrary interval I where $I \models \varphi^*$ holds, formula ψ also holds over I. If $|I| = 0$ then $I \models \psi$ follows from premise 2 of the rule and the definition of *.

Otherwise, I is chopped into several parts that satisfy ψ according to sequence ν from the definition of *. Premise 1 implies that ψ_1 holds for each part, too. Consequently, the interval satisfies $\psi_1{}^*$, according to the definition of *, and premise 2 ensures that ψ holds over I indeed. □

Since programs are just formulas in RGITL (see Section 1.5), the rule (1.1) from the introduction is just a special case of the compositionality Theorems 1.1 and 1.2 for $\odot = ;$ and $\odot = {}^*$, respectively.

To conclude, we introduce further typical operators as abbreviations:

$$\exists\, \underline{v}.\ \varphi \equiv \neg\, \forall\, \underline{v}.\ \neg\, \varphi \qquad \mathbf{E}\ \varphi \equiv \neg\ \mathbf{A}\ \neg\ \varphi \qquad \Diamond\ \varphi \equiv \mathit{true}\ \mathbf{until}\ \varphi$$
$$\Box\ \varphi \equiv \neg\ \Diamond\ \neg\ \varphi \qquad \mathbf{step} \equiv \circ\ \mathbf{last} \qquad \bullet\ \varphi \equiv \neg\ \circ\ \neg\ \varphi$$

Existential quantification $\exists\ \underline{v}$ is typically used to introduce local variables and in refinement proofs. CTL formulas ($\mathbf{E}, \mathbf{A}$) that use path quantification are sparsely used here: The allpath quantifier is used for induction, the existential path quantifier $\mathbf{E}$ is mainly used to ensure that a program does not use subformulas that are equal to *false* (see below). The eventually operator $\Diamond\ \varphi$ implies that φ holds for some postfix of an

interval. It is typically used to derive a term for well-founded induction or to specify liveness properties. The always operator $\Box\ \varphi$ states that φ holds for each postfix of an interval. It is often used to denote safety properties.[4] The **step** operator describes atomic intervals of length 1. The weak next operator states that either the interval is empty or φ holds after the first environment transition. Hence, $\bullet\ \varphi$ trivially holds in the last state of a finite interval.

1.4. Semantics of Interleaving

This section defines the compositional semantics of interleaving two formulas. From this definition, a general rule for the compositional reasoning about interleaved formulas is derived. Finally, we illustrate the semantics of the interleaving operator with a simple example.

The semantics of weak-fair $\varphi_1 \parallel \varphi_2$ and unfair $\varphi_1 \parallel_{\mathsf{nf}} \varphi_2$ interleaving, is the set of intervals that result from interleaving two *intervals* I_1 and I_2 that satisfy formulas φ_1 and φ_2, respectively.

$$
\begin{array}{ll}
I \models \varphi_1 \parallel \varphi_2 & \text{iff there are } I_1, I_2 : I_1 \models \varphi_1 \text{ and } I_2 \models \varphi_2 \text{ and } I \in I_1 \parallel I_2 \\
I \models \varphi_1 \parallel_{\mathsf{nf}} \varphi_2 & \text{iff there are } I_1, I_2 : I_1 \models \varphi_1 \text{ and } I_2 \models \varphi_2 \text{ and } I \in I_1 \parallel_{\mathsf{nf}} I_2
\end{array}
$$

This definition already ensures a compositional interleaving semantics, independently of how we actually define the semantics of interleaving two intervals (given below):

Theorem 1.3 (Compositional Interleaving Semantics)
The following compositionality rule is sound for the interleaving operators $\odot \in \{\parallel,\ \parallel_{\mathsf{nf}}\}$*:*

$$\frac{\varphi_1 \vdash \psi_1 \qquad \varphi_2 \vdash \psi_2 \qquad \psi_1 \odot \psi_2 \vdash \psi}{\varphi_1 \odot \varphi_2 \vdash \psi}$$

Proof For validity of the conclusion, we have to show that precondition $I \models \varphi_1 \odot \varphi_2$ for an arbitrary interval I, implies that I also satisfies ψ. Applying the above definition to this precondition gives intervals I_1, I_2 such that $I_1 \models \varphi_1$, $I_2 \models \varphi_2$ and $I \in I_1 \odot I_2$. Now, the first two premises imply $I_1 \models \psi_1$ and $I_2 \models \psi_2$, so $\psi_1 \odot \psi_2$ holds over I by the definition of the semantics of interleaving. Finally, the last premise implies that I satisfies ψ indeed. □

Rule (1.1) is a special case of Theorem 1.3 (substituting φ_1 and φ_2 with programs α_1 and α_2, respectively). Before we give the actual definition of interleaving two intervals $I_1 \parallel I_2$ and $I_1 \parallel_{\mathsf{nf}} I_2$, we introduce the notion of a *schedule* that dictates how two intervals are actually interleaved.

Definition 1.3 (Schedule)

[4] A formula φ is a safety formula iff an arbitrary infinite interval I satisfies φ already if each finite prefix $I_{[0..n]}$ can be extended with some interval I_0 such that $I_{[0..n]} + I_0$ satisfies φ. Without the extension interval I_0, e.g., a safety formula $\Box\ x' > x$ could not be shown in the last state of a finite prefix where x' is replaced with x.

- *A schedule* $sc = (sc(0), sc(1), \dots)$ *is a finite or infinite sequence, where each* $sc(i)$ *is either 1 or 2, indicating which component is scheduled for execution.*
- *An empty schedule is written* $()$ *and the length of a schedule is denoted as* $|sc|$.
- *The postfix of sc from* $sc(n)$ *is written* $sc_{[n..]}$.
- *A fair schedule is either finite or it changes the selected component infinitely often.*

Now we can define the weak-fair/unfair interleaving of two intervals as

$$I_1 \parallel I_2 \equiv \{I\colon \text{There is a fair schedule } sc \text{ such that } (I, sc) \in I_1 \oplus I_2\}$$
$$I_1 \parallel_{\mathsf{nf}} I_2 \equiv \{I\colon \text{There is a schedule } sc \text{ such that } (I, sc) \in I_1 \oplus I_2\}$$

where $I_1 \oplus I_2$ is the set of *scheduled interleavings* (I, sc) of a resulting interval I and schedule sc. Members of $I_1 \oplus I_2$ are finite or infinite scheduled interleavings that are constructed stepwise by considering 6 possible cases that result from attempting a transition of I_1 or I_2 where each transition is either terminating, non-blocked or blocked. The following definition gives the first three, where a transition of I_1 is attempted. The other three cases are symmetric.

Definition 1.4 (Scheduled Interleaving of First Component)
When I_1 *is executed first, the scheduled interleaving of intervals* I_1 *and* I_2 *is:*

1. *Interval* I_1 *is empty: If* $I_1(0) = I_2(0)$*, then the result is* $\{(I_2, ())\}$*, otherwise interleaving the two intervals is impossible and* $\emptyset$ *is returned.*
2. *The first transition of* I_1 *is not blocked, i.e.,* $I_1(0)_b = \mathrm{ff}$*: The transition is executed and the rest of* I_1 *is interleaved with* I_2*. The result then is:*
   $\{(I, sc)\colon I = (I_1(0), \mathrm{ff}, I_1'(0), \dots), sc(0) = 1, (I_{[1..]}, sc_{[1..]}) \in I_{1[1..]} \oplus I_2\}$.
3. *The first transition of* I_1 *is blocked, i.e.,* $I_1(0)_b = \mathrm{tt}$*: If* I_2 *is not empty and* $I_2(0)$ *equals* $I_1(0)$*, then a transition of* I_2 *is taken and the first transition of* I_1 *is consumed. The result is:* $\{(I, sc)\colon I = (I_2(0), I_2(0)_b, I_2'(0), \dots), sc(0) = 1, (I_{[1..]}, sc_{[1..]}) \in I_{1[1..]} \oplus I_{2[1..]}\}$*. Otherwise,* $\emptyset$ *is returned.*

Note that having no result in Case 3 of Definition 1.4 when I_2 is empty, merely avoids a duplicate of the first case when the second component is executed first. Moreover, the schedule in Definition 1.4 ends with the termination of either I_1 or I_2. Hence, a schedule can be shorter than the resulting interleaved interval I. Also note that Case 3 enforces that when component 1 is blocked, then component 2 is also considered for execution. This corresponds to the usual behavior of preemptive process schedulers. Hence, even with unfair schedules, our interleaving can not infinitely often execute blocked transitions of a component without ever executing the other component again.

Although the construction above does not terminate when both intervals are infinite, it is nevertheless well-defined, since all prefixes of $I_1 \oplus I_2$ of all lengths n are defined

by what is given by induction on n. One can alternatively turn Definition 1.4 into an equivalent non-recursive definition by using two auxiliary sequences $\underline{l} = (l(0), l(1), \ldots)$ and $\underline{r} = (r(0), r(1), \ldots)$ of natural numbers that model the interleaving status of each individual interval. That is, the k-th step of the recursion interleaves the two postfixes $I_{1[l(k)..]}$ and $I_{2[r(k)..]}$ from positions $l(k) \leq |I_1|$ and $r(k) \leq |I_2|$ of the two intervals. A formal non-recursive definition of the extended interleaving $(I, sc, l, r) \in I_1 \oplus I_2$ that considers these auxiliary sequences, and a mechanized proof that the recursive version follows from the non-recursive one, is given online [57].

1.4.1. Example: Interleaving Two Intervals

Consider the interleaving of the following two intervals I_1 and I_2. Each interval consists of just one program transition that is not blocked.

$$I_1 = (I_1(0), \mathrm{ff}, I_1'(0), I_1(1))$$
$$I_2 = (I_2(0), \mathrm{ff}, I_2'(0), I_2(1))$$

The schedule is $sc = (1, 2)$, i.e., the first transition of I_1 is executed first, and afterwards I_2. The interval I that results from interleaving I_1 and I_2 is thus

$$I = (I_1(0), \mathrm{ff}, I_1'(0), I_2(0), \mathrm{ff}, I_2'(0), I_1(1))$$

if the last states of both intervals are equal, i.e., $I_1(1) = I_2(1)$. Otherwise, interleaving is not possible, since the intervals do not end in a common state. As Figure 1.4 (left) shows, the interleaving instantiates the local environment transition of I_1 from $I_1'(0)$ to $I_1(1)$ with the sequence

$$(I_1'(0), I_2(0), I_2'(0), I_1(1))$$

in the result. Intuitively speaking, in an interleaving the *local* environment transitions of one component consist of alternating sequences of global environment transitions (from $I_1'(0)$ to $I_2(0)$ and from $I_2'(0)$ to $I_1(1)$, respectively) and program transitions of the other component (from $I_2(0)$ to $I_2'(0)$). Hence, possible environment assumptions of one component must be satisfied by sequences of transitions from the global environment and the other component. (This enforces several constraints on RG conditions, as we explain in Section 5.1.)

As a variation, consider the case where the first transition of the first interval is blocked, i.e., the interleaved intervals are

$$I_1 = (I_1(0), \mathrm{tt}, I_1'(0), I_1(1))$$
$$I_2 = (I_2(0), \mathrm{ff}, I_2'(0), I_2(1))$$

where $I_1(0)_b = \mathrm{tt}$ and the first transition of I_1 stutters $I_1(0) = I_1'(0)$. The schedule is $sc = (1)$, i.e., I_1 is executed first. The resulting interval is

$$I = (I_2(0), \mathrm{ff}, I_2'(0), I_2(1)) = I_2$$

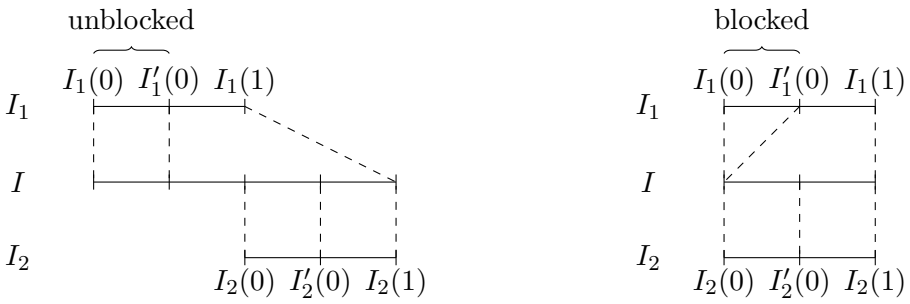


Figure 1.4.: Examples of interleaving two intervals

$$
\begin{aligned}
\alpha ::= {} & \textbf{skip} \mid \underline{z} := \underline{e} \mid \textbf{if}^* \varphi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \mid \textbf{if } \varphi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \mid \\
& \textbf{while}^* \ \varphi \textbf{ do } \alpha \mid \textbf{while } \varphi \textbf{ do } \alpha \mid \textbf{await } \varphi \mid \textbf{let } \ \underline{z} = \underline{e} \textbf{ in } \alpha \mid \\
& \textbf{choose } \ \underline{z} \textbf{ with } \varphi \textbf{ in } \alpha_1 \textbf{ ifnone } \alpha_2 \mid \\
& \alpha_1; \alpha_2 \mid \alpha^* \mid \mathtt{PROC}(\underline{e}; \underline{x}) \mid \alpha_1 \parallel \alpha_2 \mid \alpha_1 \parallel_{\mathsf{nf}} \alpha_2 \mid \\
& \varphi
\end{aligned}
$$

Figure 1.5.: Syntax of Programs in RGITL.

if $I_2(0) = I_1(0)$ and $I_2(1) = I_1(1)$. Otherwise, the interleaving is impossible. Figure 1.4 (right) shows that in this case, the first transitions of both intervals are executed according to Case 3 from Definition 1.4 and the resulting first transition of the overall interval I is not blocked, since $I_2(0)_b = \text{ff}$. The first step of I is the first step of I_2. The remaining intervals both have length zero and interleaving is possible only if $I_1(1) = I_2(1)$ according to Case 1 from Definition 1.4.

1.5. Programs

This section introduces programs in RGITL. They are mere abbreviations of specific formulas.

1.5.1. The Abstract Programming Language

Figure 1.5 defines the syntax of a program α. The program constructs have the following informal meaning: **skip** is a non-blocking stutter step. The (parallel) assignment $\underline{z} := \underline{e}$ atomically sets variables $\underline{z}$ to the current values of $\underline{e}$. The difference between $\textbf{if}^*$ and **if** (likewise for $\textbf{while}^*$ and **while**) is that the latter performs a stutter step to evaluate its test, while the former evaluates its test instantly. Thus the former version can be used to model, e.g., instructions that perform a test and an assignment atomically (in one indivisible step) such as compare-and-set (CAS). The operator **await** φ executes

a blocked transition as long as its waiting condition φ is not satisfied. It is typically used for lock-based synchronization in parallel programs with shared variables.

Local variables $\underline{z}$ for a program α can be introduced either with the **let** or the **choose** construct. Using the **let** construct, the initial values of the local variables are simply $\underline{e}$. Occasionally, we write **let** $\underline{z}$ **in** α to denote that the initial values of $\underline{z}$ are arbitrary. The **choose** construct selects arbitrary initial values that satisfy φ and executes α_1. If no such values exist, then α_2 is executed instead. The **choose** is a general construct to introduce (infinite) nondeterminism [8]. Standard nondeterminism can be derived as

$$\begin{aligned}\alpha_1 \textbf{ or } \alpha_2 \equiv \textbf{choose } B \textbf{ with } \mathit{true} \textbf{ in}\\ \{\textbf{if}^* \ B \textbf{ then } \alpha_1 \textbf{ else } \alpha_2\}\\ \textbf{ifnone skip}\end{aligned}$$

where B is a boolean variable and the **ifnone** case is irrelevant here.

The sequential composition, the star, the interleaving operators and procedure calls $\texttt{PROC}(\underline{e};\underline{x})$ are overloaded for programs and get their semantics from the corresponding operators on formula level (see below).

Finally, any formula φ can be used in general as a program according to the last clause of the grammar above. Thus we can, e.g., annotate programs with an assume statement. For instance, the program

$$(N = 1 \wedge \textbf{last}); N := N + 1$$

executes the assignment $N := N + 1$ only from states where N is 1. More importantly, the last clause of the program grammar from Figure 1.5 is typically used in compositional proofs to abstract a program by a (temporal) formula that the program satisfies (see, e.g., Section 5.1).

1.5.2. Frame Variables and Interval-Based Program Semantics

Before we actually define the formal semantics of our programming language, we extend programs with explicit frame variables. In conventional programming languages, assigning a value to a variable leaves other variables unchanged. This is different from standard ITL where other variables may change arbitrarily during an assignment. Instead, RGITL uses an explicit list of frame variables that are typically unchanged during the execution of a program (similar to TLA [63]).

Definition 1.5 (Programs with Frame Variables)
A program α with frame variables $\underline{x}$ is written $[\alpha]_{\underline{x}}$. The list $\underline{x}$ of distinct, flexible variables is called the frame assumption of α.

Note that $[\alpha]_{\underline{x}}$ is not a new type of formula but a mere syntactic abbreviation for a formula of RGITL (shallow embedding).

$$
\begin{aligned}
[\mathbf{skip}]_{\underline{x}} &\equiv \mathbf{step} \wedge \neg\, \mathbf{blocked} \wedge \underline{x'} = \underline{x} && (1.2)\\
[\underline{z} := \underline{e}]_{\underline{x}} &\equiv \underline{z'} = \underline{e} \wedge \mathbf{step} \wedge \neg\, \mathbf{blocked} \wedge \underline{y} = \underline{y'} && (1.3)\\
&\quad \text{where } \underline{y} = \underline{x} \setminus \underline{z} \\
[\mathbf{if}^*\, \varphi\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{x}} &\equiv \varphi \wedge [\alpha_1]_{\underline{x}} \vee \neg\, \varphi \wedge [\alpha_2]_{\underline{x}} && (1.4)\\
[\mathbf{if}\ \varphi\ \mathbf{then}\ \alpha_1\ \mathbf{else}\ \alpha_2]_{\underline{x}} &\equiv [\mathbf{if}^*\, \varphi\ \mathbf{then}\ (\mathbf{skip}; \alpha_1)\ \mathbf{else}\ (\mathbf{skip}; \alpha_2)]_{\underline{x}} && (1.5)\\
[\mathbf{while}^*\ \varphi\ \mathbf{do}\ \alpha]_{\underline{x}} &\equiv (\varphi \wedge [\alpha]_{\underline{x}})^*; (\neg\, \varphi \wedge \mathbf{last}) && (1.6)\\
[\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha]_{\underline{x}} &\equiv [(\mathbf{while}^*\ \varphi\ \mathbf{do}\ (\mathbf{skip}; \alpha)); \mathbf{skip}]_{\underline{x}} && (1.7)\\
[\mathbf{await}\ \varphi]_{\underline{x}} &\equiv (\neg\, \varphi \wedge \mathbf{blocked} \wedge \mathbf{step})^*; (\varphi \wedge \mathbf{last}) && (1.8)\\
[\mathbf{let}\ \underline{z} = \underline{e}\ \mathbf{in}\ \alpha]_{\underline{x}} &\equiv \exists\, \underline{y}.\ \underline{y} = \underline{e} \wedge [\alpha^{\underline{y}}_{\underline{z}}]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y'} = \underline{y''} && (1.9)\\
[\mathbf{choose}\ \underline{z}\ \mathbf{with}\ \varphi\ \mathbf{in}\ \alpha\ \mathbf{ifnone}\ \alpha_0]_{\underline{x}} &\equiv && (1.10)\\
&\quad (\exists\, \underline{y}.\ \varphi^{\underline{y}}_{\underline{z}} \wedge [\alpha^{\underline{y}}_{\underline{z}}]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y'} = \underline{y''}) \vee (\neg\, \exists\, \underline{z}.\ \varphi) \wedge [\alpha_0]_{\underline{x}}
\end{aligned}
$$

Figure 1.6.: Semantics of Programs (First 3 Lines from Grammar).

Figure 1.6 now defines the interval-based semantics of the program constructs from the first three lines of the grammar from Figure 1.5 (the rest is given below). Intuitively, a **skip** is an atomic, unblocked step that leaves all variables in the frame assumption unchanged. An assignment $[\underline{z} := \underline{e}]_{\underline{x}}$ executes atomically in one unblocked step and leaves all variables from $\underline{x}$ that do not occur in $\underline{z}$ unchanged. Depending on whether φ holds in the current state, an $\mathbf{if}^*$ rewrites to either $\varphi \wedge [\alpha_1]_{\underline{x}}$ or $\neg\, \varphi \wedge [\alpha_2]_{\underline{x}}$. An **if** takes an extra stutter step for evaluating the test. Similarly, the semantics of **while** and $\mathbf{while}^*$ is defined. The **await** φ statement blocks repeatedly as long as φ is not satisfied. This gives the program's environment a chance to unblock the program.

The definition of **let** introduces new local variables $\underline{y}$ for the variables $\underline{z}$ using existential quantification. These must be disjoint from the variables used in $\underline{e}$, α and $\underline{x}$. The variables in α are renamed to these new variables, written $\alpha^{\underline{y}}_{\underline{z}}$, and variables $\underline{y}$ are added to the frame assumption. The $\Box$-formula ensures that the introduced variables $\underline{y}$ are local indeed, i.e., they are not modified in environment transitions of α runs. The semantics of **choose** is defined similarly: It chooses some arbitrary local values $\underline{z}$ that satisfy φ, binds them to new local variables $\underline{y}$ and executes α with $\underline{z}$ renamed to $\underline{y}$.

Example 1.4 (Renaming of Local Variables)
The following simple example shows why the renaming of $\underline{z}$ by fresh variables $\underline{y}$ is required in the definition of **let** *and* **choose** *(and similarly for unfolding procedure calls as defined in* (1.11) *below):*

$$[\mathbf{let}\ N = N\ \mathbf{in}\ M := N]_{M,N}$$

The program defines a local copy of the global variable N and assigns its current value

$$
\begin{array}{rcl@{\qquad}rcl}
[\alpha_1; \alpha_2]_{\underline{x}} & \equiv & [\alpha_1]_{\underline{x}}; [\alpha_2]_{\underline{x}} & [\alpha^*]_{\underline{x}} & \equiv & ([\alpha]_{\underline{x}})^* \\
[\alpha_1 \parallel \alpha_2]_{\underline{x}} & \equiv & [\alpha_1]_{\underline{x}} \parallel [\alpha_2]_{\underline{x}} & [\alpha_1 \parallel_{\mathsf{nf}} \alpha_2]_{\underline{x}} & \equiv & [\alpha_1]_{\underline{x}} \parallel_{\mathsf{nf}} [\alpha_2]_{\underline{x}} \\
[\mathtt{PROC}(\underline{e}; \underline{z})]_{\underline{x}} & \equiv & \mathtt{PROC}(\underline{e}; \underline{z}) \wedge \Box\ \underline{y} = \underline{y'}, & [\varphi]_{\underline{x}} & \equiv & \varphi \\
 & & \text{where } \underline{y} = \underline{x} \setminus \underline{z} & & &
\end{array}
$$

Figure 1.7.: Distributivity of Frame Assumptions.

to M. Just using $\exists\ N.\ N = N \wedge [M := N]_{M,N} \wedge \Box\ N' = N''$ as semantics (without renaming), would erroneously assign M an arbitrary value.

Finally, the semantics of the remaining program constructs from the last two lines of the grammar in Figure 1.5 is already defined by the semantics of the respective operators on formula-level. This is because frame assumptions distribute over these constructs as Figure 1.7 shows.[5]

General programs that introduce arbitrary formulas φ using the last clause of the grammar from Figure 1.5 can be equivalent to *false*. However, a restricted class of *regular programs* is occasionally necessary to ensure that a program has a non-empty semantics.

Definition 1.6 (Regular Programs)
A regular program never uses the last clause of the grammar from Figure 1.5 and all its expressions are state expressions.

That is, in regular programs formulas occur only in tests as state formulas (HOL formulas without priming or temporal operators, see Definition 1.1) and the right-hand side of assignments uses state expressions only. For instance, the general program

$$N := N' + 1$$

is not regular since it uses N' which is not a state expression. The program semantically evaluates to *false*.

Proposition 1.1 (Properties of Regular Programs)

- *Regular programs have a non-empty set of intervals from any initial state.*
- *A regular program α is monotonic in its calls, i.e., for two procedures $\mathtt{PROC}_1$ and $\mathtt{PROC}_2$ with the same argument types such that $\mathtt{PROC}_1^{\mathcal{A}} \subseteq \mathtt{PROC}_2^{\mathcal{A}}$ and with α' being α where all calls to $\mathtt{PROC}_1$ are replaced with calls to $\mathtt{PROC}_2$: $\{I \colon I \models \alpha\} \subseteq \{I \colon I \models \alpha'\}$*

[5] In the future parts of this work, we refrain from writing explicit frame variables for brevity.

Definition 1.7 (Procedure Declarations)
Procedure declarations are written

$$\texttt{PROC}(\underline{x};\underline{y})\ \{\alpha\}$$

and can be mutually recursive. The declaration α is subject to two restrictions that guarantee a well-defined semantics:

1. *It only assigns to its parameters $\underline{x},\underline{y}$ and local variables introduced by* **let** *and* **choose**.
2. *It is a regular program.*

Therefore, the semantics of (recursive) procedure declarations is well-defined according to Knaster-Tarski's standard fixpoint theorem. A procedure declaration $\texttt{PROC}(\underline{x};\underline{y})\ \{\alpha\}$ yields the fixpoint equation

$$\texttt{PROC}^{\mathcal{A}} = \{(I(0)(\underline{x}), I(\underline{y})) \colon I \models [\textbf{let } \underline{z} = \underline{x} \textbf{ in } \alpha^{\underline{z}}_{\underline{x}}]_{\underline{y}}\}$$

which implies the *unfolding axiom*

$$\texttt{PROC}(\underline{e};\underline{y}) \leftrightarrow \exists\ \underline{z}.\ \underline{z} = \underline{e} \wedge [\alpha^{\underline{z}}_{\underline{x}}]_{\underline{y},\underline{z}} \wedge \Box\ \underline{z'} = \underline{z''} \tag{1.11}$$

by expanding the semantics of **let**. New local variables $\underline{z}$ with initial values $\underline{e}$ are used for the input parameters $\underline{x}$. Changes to the reference parameters $\underline{y}$ are visible outside of the procedure call. (Monotonicity and correctness of the unfolding axiom has been verified in HOL [57].)

Finally, procedure calls with reference parameters $f(\underline{e})$ where f is a function variable can be reduced to standard calls with variables only, according to

$$[\texttt{PROC}(\underline{e_0};\underline{y}, f(\underline{e}))]_{\underline{y},f} \ \equiv\ \exists\ z.\ \Box\ (z = f(\underline{e}) \wedge f' = f(\underline{e};z')) \wedge [\texttt{PROC}(\underline{e_0};\underline{y}, z)]_{\underline{y},z} \tag{1.12}$$

where the fresh flexible variable z records the values of parameter $f(\underline{e})$. The constraint $z = f(\underline{e})$ establishes that z equals $f(\underline{e})$ before each program transition and after each environment transition. Environment transitions may modify f arbitrarily, but program transitions update f at $\underline{e}$ just as the procedure modifies z.[6]

1.5.3. Specifications

A basic specification

$$SP = (SIG, F, St, Ax, Decl)$$

consists of a signature SIG, flexible/static variables F/St, a set of axioms Ax and a set of procedure declarations $Decl$. The semantics of a specification is the set of all algebras $\mathcal{A}$ in which all axioms $\varphi \in Ax$ are valid: $\mathcal{A} \models \varphi$.

[6] We define function modification $f(\underline{e};z) \equiv (\lambda\ \underline{u}.\ \text{if } \underline{u} = \underline{e} \text{ then } z \text{ else } f(\underline{u}))$ with new variables $\underline{u}$.

Axioms typically specify abstract data types such as natural numbers, arrays or lists. Specifications can fix a specific semantics $\texttt{PROC}^{\mathcal{A}}$ for a procedure `PROC` by supplying a procedure declaration (see Definition 1.7). If no declaration is specified then the procedure has a semantics according to Figure 1.3. The semantics of such an undeclared procedure can be restricted using a temporal logic formula φ that serves as a contract. The contract is typically given as an axiom $\texttt{PROC} \vdash \varphi$. When we apply this contract in a proof for a specific instance of its parameters, the axiom becomes a proof obligation as we explain below.

Large specifications are structured into a hierarchy of specifications using the usual algebraic operations such as union or enrichment (see, e.g., [84]). Generic specifications that have subspecifications as parameters are also possible. For example, lists over an ordered generic parameter type of elements can be instantiated using a signature morphism for the parameter specification to get lists of natural numbers. That is, the generic elements are instantiated with natural numbers where the generic order predicate on elements is instantiated with the usual order on natural numbers. Note that instantiation requires to prove the instantiated axioms of the parameter specification as theorems over the actual specification. In particular, instantiating a specification that contains an axiom $\texttt{PROC} \vdash \varphi$ for a procedure `PROC` and a (temporal) formula φ, typically generates a proof obligation for the instances of the procedure (the declaration that is given by the morphism) and formula φ. This proof obligation requires to show that the instance satisfies the property indeed.

2. Calculus

The central deduction principle of RGITL is the symbolic execution of (interleaved) programs and temporal formulas. In Section 1.1 we have motivated the principle and given an informal description of a symbolic execution step. Now we introduce symbolic execution formally and illustrate it with a small example in Section 2.1. Apart from symbolic execution, induction rules are necessary for the verification of formulas on infinite intervals. Section 2.2 defines these rules and illustrates them with small examples. An extra Section 2.3 is dedicated to fairness rules for our weak-fair/unfair interleaving operators. All introduced rules have been mechanically verified in HOL [57] and implemented in KIV. Similar to the previous chapter, this chapter is based on our recent articles [89, 90].

2.1. Symbolic Execution

Symbolic execution shows that a formula is valid over an arbitrary given interval I by exploring step-wise all (future) states of I. To this end, each symbolic execution step moves forward to the next unprimed state of I in two phases:

1. All formulas of a sequent are rewritten to equivalent formulas in "step form".
2. Substitutions are applied that move forward to the next unprimed state of interval I.

These two phases constitute an indivisible symbolic execution step.

First, we illustrate a symbolic execution step with a small example and then we give the formal definitions for the two phases. (Recall that in concrete formulas, we typically follow the ITL convention to write flexible variables with an uppercase letter and static variables with a lowercase letter.)

Example 2.1 (Symbolic Execution with Temporal Logic)
A symbolic execution step of the sequent

$$N = 0,\ [N := N + 2; \alpha]_N,\ \Box\ N' = N'' \vdash \circ\ N = 2$$

computes the following new sequent:

$$n_0 = 0,\ n_1 = n_0 + 2,\ [\alpha]_N,\ n_1 = N,\ \Box\ N' = N'' \vdash N = 2$$

The precondition $N = 0$ is transformed to $n_0 = 0$ where the new variable n_0 represents the value of N before the step. Executing the first assignment $N := N + 2$ results

in $n_1 = n_0 + 2$ and the remaining program is α. The new variable n_1 represents the value of N after the program transition but before the first environment transition. The environment assumption can be transformed to step form with the equivalence

$$\Box\ \varphi \leftrightarrow \varphi \wedge \bullet\ \Box\ \varphi$$

which gives for $\varphi \equiv N' = N''$ the constraint $n_1 = N$ for the first environment transition and again $\Box\ N' = N''$ for the rest of the interval.

2.1.1. Phase 1 of a Symbolic Execution Step

During phase 1 of a symbolic execution step, formulas are transformed to *step form*. A formula in step form typically consists of two parts: one that refers to the first three states of an interval and a second one that talks about the rest of the interval from the third (unprimed) state on.

Definition 2.1 (Step Form, Regular Formula)

- *A step formula τ is a formula that only refers to the first three states of an interval, i.e., it consists of constructs from the first line of the grammar in Figure 1.2 plus formulas* **last** *and* **blocked**.[1]
- *A next formula is a formula of the form $\circ\ \varphi$.*
- *A formula χ is in step form if it is a predicate logic combination of step formulas τ and next formulas $\circ\ \varphi$:*

$$\chi ::= \tau \mid \circ\ \varphi \mid \neg\ \chi \mid \chi_1 \wedge \chi_2 \mid \forall\ \underline{v}.\ \chi$$

- *A formula is regular if it meets the following restrictions:*
  - *It contains only regular programs.*
  - *Sequential composition/iteration operators and procedure calls are used in regular programs only. In particular, procedure calls must have a regular body.*
  - *Temporal operators are used at formula level only.*
  - *Equations and lambda-expressions do not use temporal operators and applications $e(\underline{e}')$ may use temporal operators only when e is a standard boolean operator.*

Definition 2.1 ensures that step/next formulas depend on the first states/rest of an interval only.

[1] An exception are static formulas that may also use temporal operators.

Proposition 2.1 (Properties of Step/Next Formulas)
For an interval I, the validity of a step/next formula $\tau/\circ\,\varphi$ depends on the subintervals $I_{[0..1]}/I_{[1..]}$ only:

$$I \models \tau \;\; \textit{iff} \;\; I_{[0..1]} \models \tau$$
$$\textit{If } I \textit{ is not empty then: } I \models \circ\,\varphi \;\; \textit{iff} \;\; I_{[1..]} \models \varphi$$

Proof The proof of the first property uses Definition 2.1 for step formula τ. The second property follows immediately by the interval semantics of the $\circ$ operator. □

With these prerequisites we can now give the main result for the first phase of a symbolic execution step:

Theorem 2.1 (Transformation to Step Form)
Any regular formula can be transformed into an equivalent formula in step form.

Proof The proof of this theorem is by induction over the structure of the given formula. First, we consider the simple cases of temporal operators (including derived operators) which follow using the following unwinding properties:

$$\varphi \;\textbf{until}\; \psi \leftrightarrow \psi \vee (\varphi \wedge \circ\,(\varphi \;\textbf{until}\; \psi))$$
$$\bullet\,\varphi \leftrightarrow \textbf{last} \vee \circ\,\varphi$$
$$\Box\,\varphi \leftrightarrow \varphi \wedge \bullet\,\Box\,\varphi$$
$$\Diamond\,\varphi \leftrightarrow \varphi \vee \circ\,\Diamond\,\varphi$$

Since the induction hypothesis ensures that subformulas φ and ψ above are already in step form, the right-hand side of the equivalences above is in step form too.

The full proof of this theorem is not in the scope of this thesis. For the case of sequential composition and interleaving, we give symbolic execution rules for formulas in Appendix A.1. For the special case of sequential programs, we will derive Hoare-style symbolic execution rules for each individual program statement in Section 2.1.5. For further details see our article [89] or refer to the online presentation [57]. □

Theorem 2.1 can be applied only to regular formulas, but symbolic execution actually works for a larger class of formulas and we occasionally use this fact. In particular, it can be used when an undeclared procedure `PROC` is specified by a regular temporal formula φ as follows: First, we rewrite an occurrence of `PROC` that we want to execute with its abstraction φ, typically by applying rule (1.1). Since the regular formula φ can be transformed to step form according to Theorem 2.1, symbolic execution of an abstraction of `PROC` is now possible.

2.1.2. Phase 2 of a Symbolic Execution Step

For formulas in step form, two substitutions $\mathcal{L}(\chi)$ and $\mathcal{N}(\chi)$ are defined. Applying these functions constitutes phase 2 of a symbolic execution step which moves to the next unprimed state of an interval. We describe these substitutions in the following:

The first substitution $\mathcal{L}(\chi)$ is for the special case of empty intervals. Hence, it replaces all next formulas with *false*, **last** with *true*, **blocked** with *false* and all flexible variables $\underline{x}$ (unprimed, primed and double primed) in step-formulas with fresh static variables $\underline{x}_0$. For quantifiers, $\mathcal{L}(\forall\ y.\ \varphi) \equiv \forall\ y_0.\ \mathcal{L}(\varphi)$, where the $\mathcal{L}$ on the right hand side additionally replaces y, y' and y'' with the new static variable y_0.

The second substitution $\mathcal{N}(\chi)$ is for non-empty intervals. It replaces all free flexible variables $\underline{x}, \underline{x'}, \underline{x''}$ in step-formulas with $\underline{x}_0, \underline{x}_1, \underline{x}$ using fresh *static* variables $\underline{x}_0$ and $\underline{x}_1$ for the old values of $\underline{x}$ and $\underline{x'}$ respectively (here we deviate from our convention to denote static variables as u). It also replaces **blocked** with a fresh static boolean variable bv, **last** with *false* and it removes the leading next operator from next formulas, *without* substituting the body. The difficult case is a quantifier over a flexible variable: $\mathcal{N}(\forall\ y.\ \varphi)$ is defined as $\forall\ y_0, y_1, y.\ \mathcal{N}(\varphi)$ where the $\mathcal{N}$ on the right hand side replaces free variables y, y', y'' in step formulas with y_0, y_1, y, again using fresh static variables y_0, y_1. Note that double primed variables y'' are now replaced with their unprimed version y. Both substitutions have no effect on static formulas within step formulas.

Finally, we can now define a symbolic execution step as follows:

Theorem 2.2 (A Symbolic Execution Step)
If all formulas in Γ *and* Δ *can be transformed to formulas* Γ_1 *and* Δ_1 *in step form, then the following rule is sound.*[2]

$$\frac{\mathcal{N}(\Gamma_1) \vdash \mathcal{N}(\Delta_1) \qquad \mathcal{L}(\Gamma_1) \vdash \mathcal{L}(\Delta_1)}{\Gamma \vdash \Delta}\ \textit{SymbExec Step} \tag{2.1}$$

The proof of this theorem is not in the scope of this work. It can be found, e.g., in our article [89] or in the online description [57].

2.1.3. Example: Symbolic Execution of a Sequential Program

To demonstrate how Theorem 2.2 can be applied in practice, we now consider the concrete goal

$$N = 0,\ [\{\textbf{let}\ M = N + 1\ \textbf{in}\ N := M\}; N := 2]_N,\ \Box\ N'' \leq N' \vdash \Box\ N \leq N'$$

which states that starting with a counter $N = 0$ the two transitions of the program will not decrease N if the environment never increases N. (Recall the convention that in concrete formulas uppercase letters denote flexible, lowercase letters static variables.)

The first phase of a symbolic execution step transforms each formula of a sequent into step form. First we consider the top-level program which is a compound. Here the **let** and the assignment are both transformed to step form using their semantic definitions (1.9)/(1.3) and the introduced $\Box$-formula is unwound which gives

$$\exists\ M.\ M = N + 1 \wedge N' = M \wedge \neg\ \textbf{blocked} \wedge \circ\ \textbf{last} \wedge M' = M'' \wedge \bullet\ \Box\ M' = M''$$

[2] The rule is also invertible, i.e., its valid conclusion implies that its premises are also valid. Invertible rules are important in practice, since their application on a provable sequent never leads to an unprovable goal.

where the $\circ$ and the $\bullet$ formula can be contracted to $\circ\,(\mathbf{last} \wedge \Box\, M' = M'')$. Substituting the result into the sequent and computing step form for the other formulas gives

$$\begin{aligned} N = 0,\ \exists\, M. \quad & M = N + 1 \wedge N' = M \wedge \neg\, \mathbf{blocked} \wedge M' = M'' \\ & \wedge \circ\, ((\mathbf{last} \wedge \Box\, M' = M''); [N := 2]_N), \\ & N'' \leq N' \wedge (\mathbf{last} \vee \circ\, \Box\, N'' \leq N') \\ \vdash\ & N \leq N' \wedge (\mathbf{last} \vee \circ\, \Box\, N \leq N') \end{aligned}$$

The sequent is now in step form.[3]

Phase two of the symbolic execution step now creates two goals (see Theorem 2.2): The goal that uses substitution $\mathcal{L}$ is trivial, since the substitution gives false for next formula $\circ\,((\mathbf{last} \wedge \Box\, M' = M''); [N := 2]_N)$. The other goal where substitution $\mathcal{N}$ is applied, replaces N, N' and N'' with n_0, n_1 and N, respectively. Similarly, M, M', M'' become m_0, m_1 and M. Formula **last** in the succedent is replaced with *false*, **blocked** with a boolean variable bv and the next operators are dropped. Therefore, the result of the symbolic execution step is

$$\begin{aligned} n_0 = 0,\ \exists\, m_0, m_1, M. \quad & m_0 = n_0 + 1 \wedge n_1 = m_0 \wedge \neg\, bv \wedge m_1 = M \\ & \wedge\, (\mathbf{last} \wedge \Box\, M' = M''); [N := 2]_N, \\ & N \leq n_1 \wedge (\mathit{false} \vee \Box\, N'' \leq N') \\ \vdash\ & n_0 \leq n_1 \wedge (\mathit{false} \vee \Box\, N \leq N') \end{aligned}$$

Predicate logic simplification now drops the existential quantifier and proves the first conjunct of the succedent $n_0 \leq n_1$. This shows that N was not illegally decreased in the first program transition. Simplifying the rest of the sequent and dropping equations for static variables that are no longer needed gives[4]

$$N \leq 1,\ M = 1,\ [N := 2]_N,\ \Box\, N'' \leq N' \vdash \Box\, N \leq N'$$

With another symbolic execution step one gets

$$n_0 \leq 1, m_0 = 1, n_1 = 2 \wedge \neg\, bv \wedge \mathbf{last}, N \leq n_1 \wedge \bullet\, \Box\, N'' \leq N' \vdash n_0 \leq n_1 \wedge \Box\, N \leq N'$$

and after a final symbolic execution step we get a trivial goal for $\mathcal{N}$ since **last** is true now, and the goal where $\mathcal{L}$ is applied just requires to prove that $n_0 \leq n_0$.

2.1.4. Example: Symbolic Execution of an Interleaved Program

Consider a simple program that interleaves two assignments for a counter N: To prove that starting with $N = 0$ in an environment that never changes N, the program never sets N to a value greater than 2

$$N = 0, [N := 1 \parallel N := 2]_N, \Box\, N' = N'' \vdash \Box\, N' \leq 2$$

[3] The implementation in KIV simplifies a program $(\mathbf{last} \wedge \ldots); N := 2$ to $N := 2$ and also drops the existential quantifier, but to explain the basic symbolic execution step, we avoid such simplifications.

[4] Note that $u = t \vdash \varphi$ is equivalent to $\vdash \varphi$ if $u \notin \mathrm{free}(\varphi)$.

we use symbolic execution as explained in Appendix A.1. This results in the following derivation tree (read bottom up).

Executing the interleaving discerns whether the left/right component is executed first and executes it. This gives two new goals according to the left/right premise of the derivation tree.

$$\frac{\dfrac{n_0 = 2 \vdash n_0 \leq 2}{N = 1, [N := 2]_N, \Box\ N' = N'' \vdash \Box\ N \leq 2} \quad \dfrac{n_0 = 1 \vdash n_0 \leq 2}{N = 2, [N := 1]_N, \Box\ N' = N'' \vdash \Box\ N \leq 2}}{N = 0, [N := 1 \parallel N := 2]_N, \Box\ N' = N'' \vdash \Box\ N' \leq 2}$$

Then the remaining assignments are executed as explained before which closes the proof.

2.1.5. Derived Rules for Sequential Programs

From the general rule for symbolic execution (see Theorem 2.2), we now derive specific rules for the verification of sequential programs, similar to dynamic logic [43] that KIV also supports [84]. In the following, we denote the step form of additional formulas Γ/Δ in the antecedent/succedent of the rules as Γ_1/Δ_1 and use the static variable bv to store the blocking information for the last program transition.

In the last state of an interval where the remaining (program) formula is **last**, symbolic execution uses the following rule

$$\frac{\mathcal{L}(\Gamma_1) \vdash \mathcal{L}(\Delta_1)}{\Gamma, \mathbf{last} \vdash \Delta}$$

The symbolic execution of a sequential program **skip**; γ uses the following rule

$$\frac{\mathcal{N}(\Gamma_1), [\gamma]_{\underline{x}}, \underline{x_0} = \underline{x_1}, \neg\ bv \vdash \mathcal{N}(\Delta_1)}{\Gamma, [\mathbf{skip}; \gamma]_{\underline{x}} \vdash \Delta}$$

where the fresh static variables $\underline{x_0}/\underline{x_1}$ store the values of $\underline{x}$ before/after the nonblocking ($\neg\ bv$) stutter step. If the rest program γ of a sequential composition $\alpha; \gamma$ is empty, we simply rewrite α to α; **last** to get the base case.

We execute the sequential program $(\underline{z} := \underline{e}); \gamma$ according to the following rule

$$\frac{\mathcal{N}(\Gamma_1), [\gamma]_{\underline{x}}, \underline{z_1} = \underline{e_0}, \underline{y_0} = \underline{y_1}, \neg\ bv \vdash \mathcal{N}(\Delta_1)}{\Gamma, [\underline{z} := \underline{e}; \gamma]_{\underline{x}} \vdash \Delta}$$

where variables with index 0/1 are fresh static variables that refer to the state before/after the assignment and $\underline{y} = \underline{x} \setminus \underline{z}$.

The **if*** statement just gives two new premises for the *current* state according to the following rule

$$\frac{\begin{array}{c}\Gamma, \varphi, [\alpha; \gamma]_{\underline{x}} \vdash \Delta \\ \Gamma, \neg\ \varphi, [\beta; \gamma]_{\underline{x}} \vdash \Delta\end{array}}{\Gamma, [(\mathbf{if}^*\ \varphi\ \mathbf{do}\ \alpha\, \mathbf{else}\, \beta); \gamma]_{\underline{x}} \vdash \Delta}$$

The **if** statement introduces a stutter step to evaluate its test[5]

$$\frac{\Gamma, \varphi, [\mathbf{skip}; \alpha; \gamma]_{\underline{x}} \vdash \Delta \quad \Gamma, \neg\, \varphi, [\mathbf{skip}; \beta; \gamma]_{\underline{x}} \vdash \Delta}{\Gamma, [(\mathbf{if}\, \varphi\, \mathbf{do}\, \alpha\, \mathbf{else}\, \beta); \gamma]_{\underline{x}} \vdash \Delta}$$

The **while*** statement gives two new premises for the current state according to the following rule

$$\frac{\Gamma, \varphi, [\alpha; ((\mathbf{while}^*\, \varphi\, \mathbf{do}\, \alpha); \gamma)]_{\underline{x}} \vdash \Delta \quad \Gamma, \neg\, \varphi, [\gamma]_{\underline{x}} \vdash \Delta}{\Gamma, [(\mathbf{while}^*\, \varphi\, \mathbf{do}\, \alpha); \gamma]_{\underline{x}} \vdash \Delta}$$

Similarly, the iteration operator * gives two new premises for the current state according to the following rule

$$\frac{\Gamma, [\alpha; (\alpha^*; \gamma)]_{\underline{x}} \vdash \Delta \quad \Gamma, [\gamma]_{\underline{x}} \vdash \Delta}{\Gamma, [\alpha^*; \gamma]_{\underline{x}} \vdash \Delta}$$

The **while** statement introduces a stutter step to evaluate its test according to the following rule

$$\frac{\Gamma, \varphi, [\mathbf{skip}; (\alpha; (\mathbf{while}\, \varphi\, \mathbf{do}\, \alpha); \gamma)]_{\underline{x}} \vdash \Delta \quad \Gamma, \neg\, \varphi, [\mathbf{skip}; \gamma]_{\underline{x}} \vdash \Delta}{\Gamma, [(\mathbf{while}\, \varphi\, \mathbf{do}\, \alpha); \gamma]_{\underline{x}} \vdash \Delta}$$

An **await** φ statement (where φ is a state formula) followed by a rest program γ executes according to the following rule

$$\frac{\Gamma, \varphi, [\gamma]_{\underline{x}} \vdash \Delta \quad \mathcal{N}(\Gamma_1 \wedge \neg\, \varphi), [\mathbf{await}\, \varphi; \gamma]_{\underline{x}}, \underline{x_0} = \underline{x_1}, bv \vdash \mathcal{N}(\Delta_1)}{\Gamma, [\mathbf{await}\, \varphi; \gamma]_{\underline{x}} \vdash \Delta}$$

In the first premise, the rest program remains to be executed and φ is added to the antecedent. In the second premise, a blocked stutter step has been executed.

The **let** statement rewrites to the following new premise for the current state

$$\frac{\Gamma, \exists\, \underline{y}.\ \underline{y} = \underline{e} \wedge [\alpha^{\underline{y}}_{\underline{z}}; \gamma]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y'} = \underline{y''} \vdash \Delta}{\Gamma, [(\mathbf{let}\, \underline{z} = \underline{e}\, \mathbf{in}\, \alpha); \gamma]_{\underline{x}} \vdash \Delta}$$

where variables $\underline{y}$ are fresh according to Definition (1.9) in Figure 1.6.

[5]The implementation in KIV instantly executes the introduced stutter step as explained above.

Similarly, the **choose** statement gives two new premises for the current state

$$\frac{\begin{array}{l}\Gamma, \exists\, \underline{y}.\ \varphi^{\underline{y}}_{\underline{z}} \wedge [\alpha^{\underline{y}}_{\underline{z}};\, \gamma]_{\underline{x},\underline{y}} \wedge \Box\ \underline{y'} = \underline{y''} \vdash \Delta \\ \Gamma, \neg\, \exists\, \underline{z}.\ \varphi, [\beta;\, \gamma]_{\underline{x}} \vdash \Delta\end{array}}{\Gamma, [(\textbf{choose}\ \underline{z}\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta);\, \gamma]_{\underline{x}} \vdash \Delta}$$

for fresh variables $\underline{y}$ as in Definition (1.10) of Figure 1.6.

A procedure call PROC with input variables $\underline{x}$ is rewritten with its implementation α in the current state

$$\frac{\Gamma, \exists\, \underline{z}.\ \underline{z} = \underline{e} \wedge [\alpha^{\underline{z}}_{\underline{x}};\, \gamma]_{\underline{y},\underline{z}} \wedge \Box\ \underline{z'} = \underline{z''} \vdash \Delta}{\Gamma, [(\texttt{PROC}(\underline{e}; \underline{y}));\, \gamma]_{\underline{x}} \vdash \Delta}$$

according to the unfolding axiom (1.11). If no implementation is given for PROC, then we can *not* symbolically execute it. Instead of providing an implementation, PROC can also be specified by a regular formula φ as $\texttt{PROC} \vdash \varphi$. Then PROC is typically rewritten to φ which is then executed by directly applying Theorem (2.2).

2.2. Induction

This section introduces the central induction principles of RGITL. First, we give an induction rule for temporal logic. Then we define a generic induction principle that extracts a counter variable for well-founded induction from a known liveness property. Finally, we briefly describe our lazy induction scheme that saves the proof engineer from defining several induction hypotheses when different goals are repeated during symbolic execution.

2.2.1. Well-Founded Induction

In higher-order logic, the inductive proof of a formula $\varphi(n)$ over a well-founded order $\prec$ uses an induction hypothesis

$$\forall\, \underline{v}, n_0.\ n_0 \prec n \rightarrow \varphi^{n_0}_n \tag{2.2}$$

where variables $\underline{v}$ are those in $\text{free}(\varphi^{n_0}_n) \setminus \{n_0\}$.

For temporal reasoning this is not enough since the induction hypothesis would only be given for the *current* interval, while de facto it holds for *all* intervals.[6] Therefore, we use a stronger induction hypothesis which also quantifies over all possible intervals using the allpath operator **A**.

Moreover, proving a formula φ by well-founded induction over a given expression e, first introduces a *fresh* static variable n that stores the current value of e in the initial

[6] This is particularly relevant when the induction hypothesis must be applied in an interleaved context that only consists of fragments of the original interval.

state of a given interval. Then applying the standard rule for induction (2.2) on the sequent $n = e \vdash \varphi$ gives the following induction rule

$$\frac{n = e, \mathit{IndHyp}(n) \vdash \varphi}{\vdash \varphi} \tag{2.3}$$

where the induction hypothesis $\mathit{IndHyp}(n)$ is

$$\mathbf{A}\ \forall\ \underline{v}.\ e \prec n \rightarrow \varphi$$

and variables $\underline{v}$ are those in $\mathrm{free}(\varphi) \cup \mathrm{free}(e)$.[7]

Since the induction hypothesis depends on the *initial* value n of e only, it becomes applicable as soon as the current value of e becomes less than n during symbolic execution. When the induction hypothesis is applied, variables $\underline{v}$ can be instantiated according to a substitution function θ. Applying the induction hypothesis typically closes a sequent when symbolic execution leads to a goal that merely repeats the one where rule (2.3) has been used. This is illustrated next with a simple example.

Example 2.2 (Well-Founded Induction)
Consider the following simple sequential program that repeatedly decrements a local counter N until it reaches 0.

$$[\textbf{while}\ N > 0\ \textbf{do}\ \{N := N - 1\}]_N, \Box\ N' = N'' \vdash \Diamond\ (\textbf{last} \wedge N = 0)$$

Applying rule (2.3) using N as expression e gives

$$\begin{aligned} & n = N,\ [\textbf{while}\ N > 0\ \textbf{do}\ \{N := N - 1\}]_N, \Box\ N' = N'', \\ & \boldsymbol{A}\ \forall\ N.\ N < n \rightarrow \\ & \qquad ([\textbf{while}\ N > 0\ \textbf{do}\ \{N := N - 1\}]_N \wedge \Box\ N' = N'' \rightarrow \Diamond\ (\textbf{last} \wedge N = 0)) \\ \vdash\ & \Diamond\ (\textbf{last} \wedge N = 0) \end{aligned}$$

where n stores the initial value of counter N and the last formula in the antecedent corresponds to $\mathit{IndHyp}(n)$ in rule (2.3) where $\underline{v}$ is just N.[8]

After symbolically executing the loop body, the initial goal is repeated (the case where N is 0 initially is trivial) and the current value of N (which is $n-1$) is now less than n:

$$\begin{aligned} & n - 1 = N,\ [\textbf{while}\ N > 0\ \textbf{do}\ \{N := N - 1\}]_N, \Box\ N' = N'', \\ & \boldsymbol{A}\ \forall\ N.\ N < n \rightarrow \\ & \qquad ([\textbf{while}\ N > 0\ \textbf{do}\ \{N := N - 1\}]_N \wedge \Box\ N' = N'' \rightarrow \Diamond\ (\textbf{last} \wedge N = 0)) \\ \vdash\ & \Diamond\ (\textbf{last} \wedge N = 0) \end{aligned}$$

Instantiating the induction hypothesis with N closes the goal.

[7] Typically, $\mathrm{free}(e) \subseteq \mathrm{free}(\varphi)$ but for a general expression e this is not necessarily the case.
[8] Induction hypotheses are typically invariant over symbolic execution steps, see [57, 89].

2.2.2. Prefix Induction with Counters

A term for well-founded induction is not always as obvious as in Example 2.2. For instance, if we want to prove

$$M = 1, [\textbf{while } \mathit{true} \textbf{ do } \{M := M + 1\}]_M, \Box\ M' = M'' \vdash \Box\ M \geq 1$$

then the only free variable M is not a useful induction term since it is never decremented by the program. For such cases, RGITL offers a generic induction rule that derives a term for well-founded induction from a known liveness property. This liveness property is given here when we try to prove the succedent formula $\Box\ M \geq 1$ by contradiction: Moving this formula from the succedent to the antecedent under negation introduces the known liveness property $\Diamond\ M < 1$. The resulting sequent is:

$$M = 1, [\textbf{while } \mathit{true} \textbf{ do } \{M := M + 1\}]_M, \Box\ M' = M'',\ \Diamond\ M < 1 \vdash \mathit{false}$$

In general, when a liveness property $\Diamond\ \varphi$ is known, then we can use the following simple rewrite rule to introduce a *fresh* counter N that serves as an expression for well-founded induction in rule (2.3):

$$\Diamond\ \varphi \leftrightarrow \exists\ N.\ N = N'' + 1\ \textbf{until}\ \varphi \tag{2.4}$$

For a given sequent where a formula $\Diamond\ \varphi$ in the antecedent has been rewritten according to (2.4) and the existential quantifier has been dropped, each symbolic execution step decrements counter N until a state is reached where φ holds. (Formula $N = N'' + 1$ is equivalent to $N > 0 \wedge N'' = N - 1$.) Intuitively, N counts the number of symbolic execution steps that are required to reach a state where φ holds. Thus it corresponds to the length of a finite prefix of the current interval that ends in a state where φ holds.

Using (2.3) and (2.4) we can thus derive the following induction rule for proving properties $\Box\ \varphi$

$$\frac{n = N,\ N = N'' + 1\ \textbf{until}\ \neg\ \varphi,\ \mathit{IndHyp}(n),\ \Gamma \vdash \Delta}{\Gamma \vdash \Delta,\ \Box\ \varphi} \tag{2.5}$$

where

$$\mathit{IndHyp}(n)\ \equiv\ \mathbf{A}\ \forall\ \underline{v}, N.\ N < n \rightarrow ((N = N'' + 1\ \textbf{until}\ \neg\ \varphi) \wedge \bigwedge \Gamma \rightarrow \bigvee \Delta)$$

and variables $\underline{v}$ are free in the rule's conclusion. After applying rule (2.5) and at least one step of symbolic execution, the counter N gets decremented and the induction hypothesis becomes applicable to close repeated goals.[9]

The implementation of rule (2.5) in KIV immediately inserts the equation $n = N$. This gives an induction hypothesis over N

$$\mathit{IndHyp}(N)\ \equiv\ \mathbf{A}\ \forall\ \underline{v}, N_0.\ N_0 < N \rightarrow ((N_0 = N_0'' + 1\ \textbf{until}\ \neg\ \varphi) \wedge \bigwedge \Gamma \rightarrow \bigvee \Delta)$$

[9] Such induction principles are special cases of induction over the length of a prefix of the current interval (called prefix induction) which is possible for safety formulas φ in general. More details on RGITL's **prefix** operator for safety induction are discussed in [6, 7].

where counter N_0 is introduced by bounded renaming. After one symbolic execution step of formula $N = N'' + 1$ **until** $\neg\ \varphi$, we typically get the equation $n_0 = N + 1$ in the antecedent where the static variable n_0 is the initial value of N. Then the induction hypothesis becomes $\mathit{IndHyp}(n_0)$ which is directly rewritten to $\mathit{IndHyp}(N + 1)$. Hence, after k steps of symbolic execution, the induction hypothesis is $\mathit{IndHyp}(N + k)$ where $N + k$ corresponds to the initial value of N as required.

Example 2.3 (Prefix Induction with Counters)
To prove that the simple program $w \equiv [\textbf{while}\ \mathit{true}\ \textbf{do}\ \{M := M + 1\}]_M$ *above, started with a local counter* $M = 1$*, never leads to a counter* M *that is less than 1*

$$M = 1, w, \Box\ M' = M'' \vdash \Box\ M \geq 1$$

we first generalize formula $M = 1$ *to the loop invariant* $M \geq 1$ *and then apply rule* (2.5)*. (The generalization is required, since otherwise the initial goal is never repeated.) After symbolically executing the loop body once, the following goal evolves*

$$\begin{array}{l} M \geq 1,\ w,\ \Box\ M' = M'', N = N'' + 1\ \textbf{until}\ M < 1, \\ \boldsymbol{A}\ \forall\ M, N_0.\ N_0 < N + 2 \rightarrow \\ \qquad ((N_0 = N_0'' + 1\ \textbf{until}\ M < 1) \wedge M \geq 1 \wedge w \wedge \Box\ M' = M'') \rightarrow \Box\ M \geq 1 \\ \vdash \Box\ M \geq 1 \end{array}$$

which is closed with the induction hypothesis, instantiating M, N_0 *with* M, N*. Note that counter* N *has been decreased twice, since one (stutter) step of symbolic execution is necessary for evaluating the loop test and the second step increments* M*.*

2.2.3. Lazy Induction

In the presence of interleaving, induction rules must be typically applied several times to cope with different repeated goals that evolve during symbolic execution. This can be reduced to applying induction only once using our lazy induction scheme that we explain next.

The basic idea is to use a higher order variable $\mathit{IH}(N)$ as placeholder for an accumulated induction hypothesis instead of having just one fixed concrete induction formula $\mathit{IndHyp}(N)$ as in rules (2.3) and (2.5). As a motivating example, consider the following initial goal g

$$g:\quad [X := e_x; (\underbrace{\textbf{while}^*\ \mathit{true}\ \textbf{do}\ \{Y := e_y; \alpha\}}_{w_1}\ \|\ \underbrace{\textbf{while}^*\ \mathit{true}\ \textbf{do}\ \{Z := e_z; \beta\}}_{w_2})] \vdash \Box\ \varphi$$

where we want to prove formula $\Box\ \varphi$ for the interleaving of two while loops w_1 and w_2 that is prefixed with an initial assignment $X := e_x$. (Frame variables are ignored here.)

Figure 2.1 shows the structure of the derivation tree for proof goal g: The solid lines represent symbolic execution steps, the dotted lines represent sequences of such

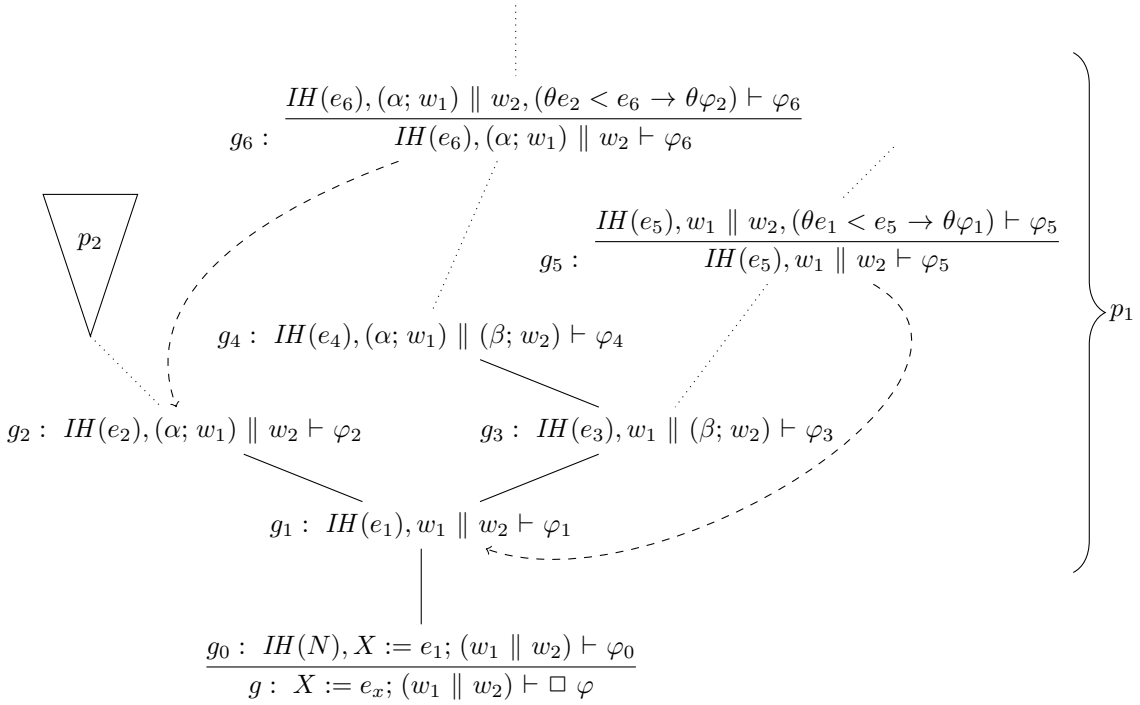


Figure 2.1.: Example: Lazy Induction

steps, proof tree p_1 designates the whole part of the tree above g_1, and proof tree p_2 corresponds to the sub-proof of g_2. Branches to the left/right denote symbolic execution steps where the left/right program is executed (some are omitted).

Similar to rule (2.5), we first introduce a counter N for well-founded induction which gives goal g_0. (For reasons of space we leave formula $N = N'' + 1\,\mathbf{until}\,\neg\,\varphi$ implicit in φ_0 here.) The meaning of the introduced lazy induction variable *IH* is explained below. Next we execute the initial assignment which leads to g_1 where the previous value of N is given by expression e_1 which is equal to $N + 1$ as explained before. Then the different possible interleavings are considered: The rightmost branch leading to goal g_5 executes the body of w_2 once, without scheduling the left program. At this point, induction must be applied to close the repeated goal. The matching goal is g_1 (depicted as a dashed arrow from g_5 to g_1), since it has the same program formula as g_5.

To deal with the repeated proof obligation g_5, we would have to apply the induction rule (2.5) for goal g_1. Similarly, if the left program is executed in g_3, then goal g_6 is eventually reached for which goal g_1 does not provide a helpful induction hypothesis since the program formulas differ. The right induction hypothesis can be found in a *different branch* of the derivation tree where only the first assignment of w_1 has been executed, according to goal g_2. Hence, the induction hypothesis must be *generalized* to consider (at least) both g_1 and g_2.

To avoid such manual inductive generalizations, our lazy induction rule introduces a

higher order variable $IH(N)$ that serves as a placeholder for a generalized induction hypothesis. Moreover, $IH(N)$ keeps track of the initial value of counter N that increases with every step of symbolic execution. Intuitively speaking, variable $IH(N + k)$ provides an induction hypothesis for *all* proof goals that evolve from executing more than k steps since introducing $IH(N)$. The benefit of this technique is that IH never needs to be defined explicitly. Only when we apply lazy induction to close a current goal, a suitable induction hypothesis is added to the current sequent. The following picture illustrates this:

$$\frac{IH(e_1), \theta e_2 \prec e_1 \rightarrow \theta\varphi_2 \vdash \varphi_1}{IH(e_1) \vdash \varphi_1} \qquad\qquad IH(e_2) \vdash \varphi_2$$

The premise on the left-hand side shows the goal after applying lazy induction and the right-hand side represents the goal that is used to generate the induction hypothesis for a given substitution θ.

This lazy induction scheme is sound, since a traditional proof that uses only well-founded induction can always be constructed from one that uses lazy induction. This construction is not in the scope of this work. It is described in our journal article [89].

2.3. Fairness

Reasoning about an interleaved program $\alpha_1 \parallel \alpha_2$ using symbolic execution is indifferent to whether the interleaving is unfair or weak-fair. This is because symbolic execution only explores finite interval prefixes, while fairness is relevant on infinite intervals only. Hence, symbolic execution alone is not sufficient to express fairness and we need another way to ensure that, e.g., in a weak-fair interleaving "each of the interleaved components is eventually scheduled for execution". This is achieved in terms of fairness rules for our two interleaving operators which we introduce in the following. All introduced fairness rules have been mechanically verified correct in HOL [57].

To formalize fairness, we extend our interleaving operators with so called scheduling labels $L_1\colon \alpha_1 \parallel L_2\colon \alpha_2$ (and similarly for $\parallel_{\mathsf{nf}}$), where L_1 and L_2 are two formulas that enforce a step of α_1 or α_2, respectively. Intuitively, whenever label L_1 is true in the current state, the next program transition of the interleaving must be one of α_1. If α_1 is currently blocked, then a transition of α_2 is also executed. If α_1 terminates, then L_1 has no effect. Unlabeled interleaving is thus a special case of labeled interleaving where both labels are false. For a formal semantics of the extended interleaving operators see [57, 89].

Using scheduling labels we can define the following central fairness rule for *weak-fair* interleaving

$$\alpha_1 \parallel \alpha_2 \leftrightarrow \exists\, B.\ \Diamond\, B \wedge (B\colon \alpha_1 \parallel \alpha_2) \tag{2.6}$$

which states that there exists a number of steps after which the new boolean variable B becomes true. Since B is also a scheduling label for the first component, this implies that α_1 must eventually execute a transition.

Example 2.4 (Exploiting Fairness for $\|$)
A simple example demonstrates the interplay between the given rules. The proof obligation

$$X = 0, [X := 1 \parallel \mathbf{skip}^*]_X,\ \Box\ X' = X'' \vdash \Diamond\ X = 1$$

can be derived by applying rule (2.6) *and then rule* (2.4) *on* $\Diamond\ B$ *to introduce a counter variable* N *for well-founded induction. Applying induction rule* (2.3) *over* N *then yields*

$$\begin{aligned} &X = 0,\ [B\text{: } X := 1 \parallel \mathbf{skip}^*]_X,\ \Box\ X' = X'', \mathit{IndHyp}(N), N = N'' + 1\ \mathbf{until}\ B \\ &\vdash \Diamond\ X = 1 \end{aligned}$$

A symbolic execution step of the introduced **until** *formula in the antecedent now discerns whether* B *is true in the current state: If this is the case, then the left process is executed and the proof goal is discarded since we then reach a state where* $X = 1$ *holds. Otherwise, when the second component executes a stutter step, the proof goal is repeated and (since* N *decreases) can be discarded with the induction hypothesis.*

For unfair interleaving $\alpha_1 \parallel_{\mathsf{nf}} \alpha_2$, a similar fairness rule holds: Either α_1 is eventually executed after some steps, just as in the case for weak-fair interleaving, or the scheduling considers some infinite nonblocking execution of α_2 only:

$$\begin{aligned} \alpha_1 \parallel_{\mathsf{nf}} \alpha_2\ \leftrightarrow\ &(\exists\ B.\ \Diamond\ B \wedge (B\text{: } \alpha_1 \parallel_{\mathsf{nf}} \alpha_2)) \\ &\vee\ \alpha_2 \wedge \Box\ \neg\ \mathbf{last} \wedge \Box\ \neg\ \mathbf{blocked} \wedge \mathbf{E}\ \exists\ \underline{x}.\ \alpha_1 \end{aligned} \tag{2.7}$$

Compared with weak-fair interleaving, rule (2.7) introduces only a simple additional case according to the second disjunct above. Note that when α_2 terminates or executes a blocked stutter step, the scheduler would execute α_1 according to Cases 1/3 of our interleaving semantics (see Definition 1.4). Formula $\mathbf{E}\ \exists\ \underline{x}.\ \alpha_1$ where $\underline{x}$ are free in α_1, ensures that α_1 is not equivalent to *false*, since otherwise the interleaving would have an empty semantics.

A second fairness rule is often required when an unfairly interleaved component satisfies a local eventually property $\Diamond\ \varphi$. The rule introduces an auxiliary counter variable N for well-founded induction

$$\begin{aligned} (\Diamond\ \varphi \wedge \alpha_1) \parallel_{\mathsf{nf}} \alpha_2 \leftrightarrow \exists\ N.\quad &\Box\ (N = N' \wedge N'' \leq N') \\ &\wedge\ ((N' = N'' + 1\,\mathbf{until}\,\varphi) \wedge \alpha_1) \parallel_{\mathsf{nf}} (\alpha_2 \wedge \Box\ N'' \leq N') \end{aligned} \tag{2.8}$$

that is decremented by one in the local environment steps of α_1 until φ holds locally. The overall program/environment transitions leave N unchanged/never increment N.

For the special case where α_1 *never blocks*, we introduce the following rule

$$\begin{aligned} &(\Diamond\ \varphi \wedge \alpha_1) \parallel_{\mathsf{nf}} \alpha_2 \leftrightarrow \exists\ N. \\ &\quad \Box\ N' = N'' \wedge ((N = N' + 1\,\mathbf{until}\,\varphi \wedge \alpha_1) \parallel_{\mathsf{nf}} (\alpha_2 \wedge \Box\ N = N')) \end{aligned} \tag{2.9}$$

that is slightly simpler to use than rule (2.8). Intuitively, the rule introduces a fresh counter N that decreases in program transitions of component 1 until φ holds locally. The other component as well as the system's environment leave N constant.[10]

Finally, the following symmetric versions of rules (2.7), (2.8) and (2.9) also hold

$$\begin{aligned}
\alpha_1 \parallel_{\mathsf{nf}} \alpha_2 \leftrightarrow \quad & (\exists\, B.\ \Diamond\, B \wedge (\alpha_1 \parallel_{\mathsf{nf}} B\colon \alpha_2)) \\
& \vee\ \alpha_1 \wedge \Box\, (\neg\, \mathbf{last} \wedge \neg\, \mathbf{blocked}) \wedge \mathbf{E}\, \exists\, \underline{x}.\ \alpha_2 \text{ with } \underline{x} \text{ free in } \alpha_2 \\
\alpha_1 \parallel_{\mathsf{nf}} (\Diamond\, \varphi \wedge \alpha_2) \leftrightarrow \exists\, N. \quad & \Box\, (N = N' \wedge N'' \leq N') \\
& \wedge\ (\alpha_1 \wedge \Box\, N'' \leq N') \parallel_{\mathsf{nf}} ((N' = N'' + 1\, \mathbf{until}\, \varphi) \wedge \alpha_2) \\
\alpha_1 \parallel_{\mathsf{nf}} (\Diamond\, \varphi \wedge \alpha_2) \leftrightarrow \exists\, N. \quad & \Box\, N' = N'' \\
& \wedge\ ((\alpha_1 \wedge \Box\, N = N') \parallel_{\mathsf{nf}} (N = N' + 1\, \mathbf{until}\, \varphi \wedge \alpha_2))
\end{aligned}$$

We use fairness rules (2.7) and (2.8) in the soundness proofs of our RG rules (see Section 5.1). Fairness rule (2.9) is used in the soundness proof of our compositional proof method for lock-freedom that is relevant for nonblocking algorithms (see Section 11.3).

[10] This rule can not be used with blocking steps of α_1, since a blocked transition leaves *all* variables unchanged. The nice trick to avoid this technical problem if necessary, is to let the environment decrement the auxiliary counter, as in rule (2.8).

3. Related Work and Conclusion

Comparison with Own Previous Work

RGITL is mainly based on the Ph.D. thesis [5] that already specifies and implements the essential parts of the logic. It has been further developed as described in the Ph.D. thesis [6]. Compared to [5,6], further improvements have been made during this thesis as follows:

- One substantial effort of several person months has been to mechanically specify and verify the semantic foundation of RGITL using higher-order logic. These results are presented online [57] and have led to various minor clarifications and improvements both of the logic and its implementation.
- In [5, 6], symbolic execution is based on Theorem 2.2 only. In contrast, we now use specific rules for the symbolic execution of sequential programs as described in Section 2.1.5. This approach leads to more intuitive proofs that are close to proofs for sequential programs using dynamic logic which KIV supports too.
- The use of step form according to Definition 2.1 instead of just the "normal form" in [5,6] leads to a more efficient symbolic execution of top-level formulas in KIV.
- Our lazy induction scheme improves the rule for repeated induction in [5] and leads to more convenient inductive proofs (see Section 2.2).
- Driven by our main application domain of concurrent data structures, we have defined new operators and rules in RGITL. For instance, the unfair interleaving operator $\|_{\mathsf{nf}}$ was required for the correct specification/verification of lock-freedom (see Section 11.3). Therefore, we have specified/verified and implemented new fairness rules for this operator (see Section 2.3). Similarly, we have introduced new operators for RG reasoning that will be described in Section 4.2.

Comparison with Other Approaches

RGITL incorporates several ideas from ITL [74] such as having both finite and infinite intervals in the semantics, or having programs as formulas with compositional operators ; and * for sequential programs. Considering finite intervals is beneficial for the specification of termination ($\Diamond$ **last**), which can be easily distinguished from infinite stuttering. Different from ITL, our interval semantics has explicit blocked program transitions to directly express the absence of deadlock as $\Box \neg$ **blocked**. (We will introduce a compositional proof method for this property in Section 5.3.)

Our semantics does not have built-in stuttering such as in [63]. This is beneficial, since we do not have to distinguish whether a formula is stutter-invariant. However, refinement proofs typically must add stuttering transitions $\mathbf{skip}^*$ explicitly to a program (see Section 10.1). We encode fairness using specific auxiliary variables (similar to [4]) that are introduced whenever this is required. In contrast, other approaches like [63] encode fairness using additional temporal logic constraints.

Different from ITL, our interval semantics has built-in environment transitions – similar to reactive sequences in [86] – where each environment transition represents sequences of transitions of the environment as *one* abstract transition. While finite reactive sequences always end with a program transition, our finite (non-empty) intervals end in an environment transition. Our interval semantics also differs from the Aczel-trace semantics in [86], which records each environment transition explicitly. In RGITL, sequences of explicit transitions of other interleaved components (and the overall environment) are visible only during the symbolic execution of an interleaving. This is beneficial for the symbolic execution of sequential programs, since it does not have to take the actual number of environment transitions into account. Moreover, ruling out infinite sequences of environment steps with fairness assumptions is unnecessary for sequential programs in our setting. For the compositional verification of some properties, however, it can be unfavorable not to be able to talk about the intermediate states that evolve during several environment steps.

Different from other approaches [27, 63, 74], we use a basic interleaving operator $\|$, similar to [86, 109] rather than deriving parallel composition from the conjunction of the behaviors of the constituent programs. From a theoretical point of view, having a non-trivial interleaving operator as a basic operator of the logic can make a completeness proof more challenging. From a practical point of view, however, an explicit interleaving operator is the right choice for verifying concurrent *programs* with shared resources rather than, e.g., physical devices where the aspect of "true parallelism" can be of greater importance and conjunction may be the better choice.

As an example, consider the following simple program $N := 1 \parallel N := 2$ that executes two reads of a shared counter N in parallel (see Section 2.1.4): With a parallel composition operator that corresponds to conjunction, the semantics of this program is equivalent to *false*, while the program *has* valid executions on a real (multi-core) machine. Instead of introducing non-atomic assignments as in [27, 75], we prefer to stick to assignments as atomic instructions which is in accordance with most real processors that support various atomic instructions such as read/write, fetch-and-increment or compare-and-swap. Of course, using atomic assignments like $M := N + 1$ where both M and N are shared variables is not realistic. Therefore, we typically require assignments and tests in conditionals to reference at most one shared variable. This corresponds to the LCR restriction from [66] which ensures that an interleaving semantics is appropriate even on multi-core processors where constituent subprograms can execute with "true parallelism".

On the other hand, if we want to consider non-atomic instructions then we can use temporal contracts for undeclared procedures that model the non-atomic behaviors of these instructions. For instance, we specify non-atomic read/write procedures in

Section 12.2. Symbolically executing these contracts typically makes the verification more involved, as it has to deal with executions that take an arbitrary (finite) number of steps and only return a valid result if no interference from concurrent processes occurs.

Conclusion

In the first part of this work, we have introduced the logic RGITL for the compositional verification of concurrent programs with shared resources. Its main features are compositional reasoning about temporal properties of sequential and interleaved programs using symbolic execution with induction. Thus we can specify/verify "domain-specific" proof methods in the logic or directly verify concrete sequential/concurrent programs correct. The second and third part of this work use these features of the logic to derive specific rules for the compositional verification of concrete interleaved programs and to verify concrete programs correct.

The semantic foundations of RGITL have been mechanically specified/verified in HOL [57] which has led to several minor improvements. For instance, when we verified the correctness of our new fairness rule (2.9) in KIV, we noticed that the rule is only sound if the first component never blocks (since no variable can be decremented in a blocked transition). While this restriction was unproblematic for our main application domain of non-blocking data structures (see Section 11.1), the rule could not be applied for the more general case with blocking behaviors. E.g., we could not prove correctness of a RG rule for total correctness of an unfairly interleaved concurrent system (see discussion at the end of Section 5.1). Thus we had to introduce a more expressive rule (2.8), i.e., we had to come up with the rule, specify/verify and implement it. Such iterative steps towards simple and expressive derivation rules constitute the main effort to improve the logic.

A completeness result for RGITL remains future work: A general rule to introduce auxiliary variables with a specific initial value, a specific environment behavior and that are "auxiliary" in program transitions is sometimes required. Such variables are currently introduced with specific fairness rules, or for the simple case of concrete programs, we simply add them manually to the program. Another open issue is finding good rules for refinement proofs $\texttt{PROC} \vdash \exists\, AS.\ \varphi$, since formula $\exists\, AS.\ \varphi$ is not a safety formula in general, so no direct inductive arguments are possible. While we typically can prove such formulas using the prefix operator [6], simple rules for symbolically executing general refinement formulas are not yet supported. Finally, it could also be beneficial to study the relationship between our interleaving operator and similar ITL operators such as projection [75].

Part II.

Rely-Guarantee Reasoning in RGITL

In the first part of this work, we have introduced the syntax and semantics of RGITL and its central deduction principles for the verification of sequential/interleaved programs and temporal formulas. In the rest of this work, we focus on various applications of RGITL. The following proof rules (theorems) are thus *derived* in the logic using the deduction principles from the first part, but they are not basic rules of the logic. Hence, the correctness proofs of these rules are not carried out on a semantic level but on calculus level.

In particular, we focus in the second part on an important class of *compositional* temporal formulas, so called rely-guarantee (RG) assertions. We derive rules that decompose RG assertions for an interleaved program to RG assertions for the sequential subprograms that are verified in a common (Hoare-style) way. Thus RG reasoning is a practical method for the specification and verification of properties of concrete concurrent algorithms and we illustrate its application using several standard examples from the literature. Both the soundness proofs of the rules and their applications are mechanized in KIV. The basic ideas described here can also be found in our articles, e.g., [89, 99]. Here we give a full description of our RG calculus (including rules for sequential programs with pre-/post-conditions) based on our new syntax for RG assertions. We have submitted a corresponding journal-article [100].

The structure of the second part of this work is as follows: Chapter 4 introduces RG assertions and several simple rules for the verification of RG assertions for sequential programs. Then we derive RG decomposition rules for programs that interleave two components in Chapter 5. Chapter 6 generalizes these parallel decomposition rules to the case of an arbitrary finite number of interleaved components. Moreover, we derive state-local RG rules that reduce the specification/verification of a concurrent system with an arbitrary number of local states to the case of just a few representative local states. Finally, Chapter 7 concludes with a discussion of related and possible future work.

4. RG Reasoning

This chapter introduces RG reasoning [53, 109] in RGITL. Section 4.1 motivates the basic ideas with the simple parallel FIND algorithm from Figure 1.1. Then Section 4.2 introduces RG assertions for partial/total correctness as a specific class of (compositional) temporal formulas and Section 4.3 explains their symbolic execution for sequential programs based on Hoare-style calculus rules.

4.1. Introduction

Compositional reasoning about parallel programs is not as simple as directly applying rule (1.1). There is more work to be done: When verifying an individual component (process) the possible interference from the other processes and the overall system's environment must be taken care of. To better understand this, reconsider the `FIND` algorithm from Figure 1.1 that decomposes the search of a minimal index in an array to two parallel search operations $\texttt{FIND}_E$ and $\texttt{FIND}_O$ for the search of even and odd indices, respectively:

One could not prove the $\texttt{FIND}_E$ operation correct under too permissive assumptions about its environment, e.g., if its environment were allowed to arbitrarily change the size of the array. Similarly, making too restrictive assumptions on the environment behavior could prevent composing the verification of an individual process into the execution context of the other process (and the global environment). E.g., assuming no concurrent changes to the program state at all for the verification of $\texttt{FIND}_E$ would be a too strong assumption as the proof could not be composed in parallel with the verification of operation $\texttt{FIND}_O$ that obviously does change the state.

RG reasoning [53] is a specification and verification technique that defines the relationship between individual and global environment assumptions and the properties that are guaranteed by components such that these can be consistently composed to yield an overall system property. It extends Hoare's well-known approach to reason about sequential programs with pre- and post-conditions to a concurrent setting as follows:

Assumptions of a process $p\colon nat$[1] about possible environment steps are specified using an extra two-state predicate $R_p\colon state \times state \to bool$ over the entire program state of type *state*. These are called rely conditions. In return, each process p must specify guarantees for its steps using a further two-state predicate $G_p\colon state \times state \to bool$, called guarantee conditions.

[1] Process identifiers are typically naturals $nat := \mathbb{N}_0$, only for FIND they are either E or O here.

For example, the following rely condition is necessary to ensure the functional correctness of the $\texttt{FIND}_E$ operation:

- R_E: Steps from the environment of process E change neither the array Ar nor its result Out_E.

Similarly, the following guarantee condition ensures that the verification of $\texttt{FIND}_O$ is compatible with the proof for $\texttt{FIND}_E$:

- G_O: Process O guarantees not to change the array Ar nor the result Out_E of process E in its own steps.[2]

In general, predicates R_p and G_p must satisfy certain criteria to ensure that they can be composed to an overall system property: Roughly speaking, the central predicate logic relation between rely and guarantee conditions is that the guarantee of one process must imply the rely conditions of each other process. The central temporal logic RG requirement is that each individual process sustains its own guarantee conditions in its steps as long as its rely conditions are not previously violated by its environment.

So for the FIND algorithm it is necessary to prove the following RG assertion for operation $\texttt{FIND}_E$ (and symmetrically for $\texttt{FIND}_O$):

> Each step of $\texttt{FIND}_E$ satisfies its specific guarantee conditions G_E as long as its environment does not violate its individual rely conditions R_E.

This type of assertion is *compositional*, i.e., under some simple predicate logic side conditions on the used predicates, the parallel composition of such individual assertions yields a global RG assertion for an overall concurrent system.

4.2. RG Assertions for Partial/Total Correctness

In RGITL, we express RG assertions as the following special type of program assertion:

$$\texttt{PROC}(S) \vdash R(S', S'') \xrightarrow{+} G(S, S') \tag{4.1}$$

In (4.1), the undeclared procedure PROC has one reference parameter variable S: *state* which represents the state of some concurrent system. Moreover, there are two unspecified binary predicates G and R over sort *state* for rely and guarantee conditions. Tacitly, we assume from now on that G/R are predicates that talk about the first program/environment transition w.r.t. some state vector variable S.

The temporal sustains operator $\xrightarrow{+}$ in (4.1) ensures that the guarantee conditions G are sustained by PROC's program transitions, as long as *previous* environment transitions have preserved its rely conditions R. In particular, the first program transition

[2] We give the complete RG instances for the FIND algorithm (including pre- and post-conditions and single-state invariants) in Section 5.2.

must satisfy G. Formally, the semantics of the sustains operator is directly derived from the **until** operator as follows:[3]

$$R \xrightarrow{+} G \;\equiv \neg\,(R \;\textbf{until}\; \neg\, G)$$

Example 4.1 (Intervals Satisfying $\xrightarrow{+}$)
In the example interval below, property $R \xrightarrow{+} G$ enforces that the program step from $I(k)$ to $I'(k)$ must satisfy G (denoted as $\Rightarrow \in G$) since R has not been violated in any previous environment transition.

$$I(0) \underset{\in \mathrm{G}}{\text{——}} I'(0) \underset{\in \mathrm{R}}{\cdots\cdots} I(1) \underset{\in \mathrm{G}}{\text{——}} \;\ldots\; \underset{\in \mathrm{G}}{\text{——}} I'(k-1) \underset{\in \mathrm{R}}{\cdots\cdots} I(k) \underset{\Rightarrow \in \mathrm{G}}{\text{——}} I'(k) \;\cdots$$

The property "if all environment steps satisfy R, then all program steps satisfy G" is in general too weak to avoid circular dependencies between system components. It is easy to see that this property is strictly weaker than $\xrightarrow{+}$ (by induction over $\Box\, G$), i.e.,

$$(R \xrightarrow{+} G) \rightarrow (\Box\, R \rightarrow \Box\, G)$$

holds, but the reverse direction does not hold as the following one-step interval shows:[4]

$$I(0) \underset{\notin \mathrm{G}}{\text{——}} I'(0) \underset{\notin \mathrm{R}}{\cdots\cdots} I(1)$$

RGITL offers native support for this important class of rely-guarantee assertions using two additional operators that are directly derived from $\xrightarrow{+}$. The syntax of these operators is influenced by dynamic logic expressions for partial and total correctness [43]:

Definition 4.1 (RG Assertions for Partial Correctness)
For state formulas Pre, Post and Inv

$$\mathit{Pre}(s,S) \vdash [R(S',S''), G(S,S'), \mathit{Inv}(S), \texttt{PROC}(S)]\; \mathit{Post}(s,S)$$

abbreviates the sequent

$$\mathit{Pre}, \texttt{PROC} \vdash ((R \wedge (\mathit{Inv}' \rightarrow \mathit{Inv}'')) \xrightarrow{+} (\textbf{if last then } \mathit{Post} \textbf{ else } (G \wedge (\mathit{Inv} \rightarrow \mathit{Inv}'))))$$

[3] Equivalent, but more complex definitions of $\xrightarrow{+}$ are also possible, e.g., using the weak until operator of TL (unless), as G **unless** $(G \wedge \neg\, R)$, or using chop and star, as $(G \wedge R \wedge \textbf{step})^*$; $(G \wedge (\neg\, \textbf{last} \rightarrow \neg\, R))$.

[4] The use of an operator similar to $\xrightarrow{+}$ can already be found in [53, 71], and also in [1].

The operator [.] thus encodes an RG assertion for partial correctness of procedure PROC. It requires that final states of PROC satisfy the two-state post condition $Post\colon state \times state \to bool$ if the procedure starts in a state where the precondition $Pre\colon state \times state \to bool$ holds. Moreover, both program and environment transitions preserve the single-state invariant $Inv\colon state \to bool$. Dropping variables S, we typically write Inv' and Inv'' to abbreviate $Inv(S')$ and $Inv(S'')$.

In some cases (e.g., in our parallel FIND example) it is beneficial to also refer to the initial overall system state in the pre- and post-conditions. Therefore, these predicates are two-state predicates where the first parameter is the initial overall system state (given as a static variable s) and the other one is the current state S. [5] However, we tacitly also use these predicates as mere single-state predicates for the current state when no reference to the initial system state is necessary. Moreover, we assume that formulas *Pre*, *Post* and *Inv* are state formulas from now on.

The second derived operator $\langle\ .\ \rangle$ defines RG assertions for total correctness.

Definition 4.2 (RG Assertions for Total Correctness)

$$Pre(s,S) \vdash \langle R(S',S''), G(S,S'), Inv(S), \texttt{PROC}(S)\rangle\ Post(s,S)$$

abbreviates the sequent

$$\begin{aligned} & Pre, \texttt{PROC} \\ \vdash\quad & ((R \wedge (Inv' \to Inv'')) \xrightarrow{+} (\textbf{if last then } Post \textbf{ else } (G \wedge (Inv \to Inv')))) \\ & \wedge (\Box\ (R \wedge (Inv' \to Inv'')) \to \Diamond\ \textbf{last}) \end{aligned}$$

This formula strengthens the previous one by adding termination (so it gives a total correctness assertion) if all environment transitions are rely transitions that preserve the invariant. We occasionally refer to the first/second conjunct in the definition of $\langle\ .\ \rangle$ as the safety/liveness part of the assertion. Moreover, we use the syntax $[\langle\ .\ \rangle]$ to denote an RG assertion for either partial or total correctness. Note that both types of RG assertions merely *propagate* the invariant *Inv*, i.e., to derive that the invariant predicate always holds it must hold in the initial overall system state.

The special case

$$Pre \vdash [R, G, true, \texttt{PROC}]\ Post$$

with a trivial invariant corresponds to the Hoare-style 5-tuple

$$\texttt{PROC}\ \underline{\textbf{sat}}\ \{Pre, R, G, Post\}$$

that is used in the literature [106, 109]. We have added an invariant predicate to our operators, since a significant part of rely and guarantee predicates often consists of preserving state-invariant properties.

RG assertions with an empty environment $\Box\ S' = S''$ and trivial RG conditions

$$Pre, \Box\ S' = S'' \vdash [true, true, true, \texttt{PROC}]\ Post$$

correspond to the classic Hoare triple $\{Pre\}$ PROC $\{Post\}$ for sequential programs.

[5] In pre-conditions we sometimes refer to the initial system state, since a program can be started at a time where concurrent processes have already altered the system state.

4.3. Executing Sequential RG Assertions

In this section we explain the symbolic execution of RG assertions for sequential programs. The verification of RG assertions for partial and total correctness uses the same rules, since symbolic execution can only explore finite prefixes of a given interval. For the case of verifying RG assertions for partial correctness (which are safety formulas) on infinite intervals, we give a general induction principle. Finally, we illustrate our approach by verifying total correctness of a simple sequential program.

4.3.1. General Symbolic Execution Scheme for RG Assertions

The symbolic execution (verification) of RG assertions for partial/total correctness can be directly derived from their definition as specific **until** formulas. In particular, the symbolic execution of the underlying sustains operator $\xrightarrow{+}$ uses the following unwinding rule

$$(R \xrightarrow{+} G) \;\leftrightarrow\; G \wedge (R \rightarrow \bullet\, (R \xrightarrow{+} G)) \tag{4.2}$$

which requires to prove G for the current program transition and $\xrightarrow{+}$ for the rest of an interval if R holds in the first environment transition.

This leads to the following general scheme for the symbolic execution of RG assertions for both partial/total correctness of a program α:

Definition 4.3 (General Scheme for Executing RG Assertions)

$$\frac{\begin{array}{ll} i) & \mathit{Pre}(s_0), \mathit{Inv}(s_0), \psi(s_0,s_1) \vdash G(s_0,s_1) \\ ii) & \mathit{Pre}(s_0), \mathit{Inv}(s_0), \psi(s_0,s_1) \vdash \mathit{Inv}(s_1) \\ iii) & \psi(s_1), R(s_1,S), \mathit{Inv}(S) \vdash [\langle R, G, \mathit{Inv}, \alpha' \rangle]\ \mathit{Post} \end{array}}{\mathit{Pre}(S), \mathit{Inv}(S) \vdash [\langle R(S',S''), G(S,S'), \mathit{Inv}(S), \alpha(S) \rangle]\ \mathit{Post}(S)}$$

Starting in some state S that satisfies some pre-condition Pre and the invariant $\mathit{Inv}(S)$, a symbolic execution step generates three proof obligations: Two predicate logic side goals *i*) and *ii*) where the guarantee and propagation of the invariant must be shown for the first α-transition from s_0 to s_1 which is given here by formula $\psi(s_0, s_1)$. In the main goal *iii*) the proof must be continued for the rest of the program α' given the strongest post-condition $\psi(s_1)$ that has been computed from pre-condition $\mathit{Pre}(s_0)$ for the first program transition $\psi(s_0, s_1)$, the rely $R(s_1, S)$ for the first environment transition plus the invariant again for the current state.[6]

It is important in premise *iii*) to only propagate the essential information from $\psi(s_1)$ over the rely transition $R(s_1, S)$ when computing the new pre-condition $\mathit{Pre}_1(S)$ for the next step. This propagation of relevant information is an interactive application-specific step in our current approach. In practice, propagation is realized by a combination of automatically applied forwarding lemmas (which add information), rewriting

[6] The old guarantee $G(s_0, s_1)$ and the invariants $\mathit{Inv}(s_0)$ and $\mathit{Inv}(s_1)$ could be added to the antecedent of iii), but this is rarely required in applications, so we prefer to implicitly weaken this historic information here.

lemmas (which simplify the new information to get $Pre_1(S)$) and weakening heuristics (which finally discard old information, e.g., $R(s_1, S)$ from the sequent). These lemmas are typically trivial and can be verified automatically, but some knowledge about KIV's rewriting system is necessary to get them right.

4.3.2. Executing RG Assertions for Sequential Programs

From the general symbolic execution scheme in Definition 4.3, we now derive Hoare-style rules for the symbolic execution of RG assertions for sequential programs. (Afterwards, we illustrate the use of these rules to prove total correctness of a simple program.) For brevity, we omit additional formulas Γ/Δ in the antecedent/succedent of the following rules.

When symbolic execution reaches the last state of an interval, i.e., the remaining program formula is **last**, then the following base rule

$$\frac{Pre(S), Inv(S) \vdash Post(S)}{Pre(S), Inv(S) \vdash [\langle R(S', S''), G(S, S'), Inv(S), \mathbf{last}\rangle]\ Post(S)}$$

simply rewrites an RG assertion to its post-condition.

The symbolic execution of a sequential program **skip**; γ uses to the following rule

$$\frac{Pre(s_0), R(s_0, S), Inv(S), \neg\ bv \vdash [\langle R(S', S''), G(S, S'), Inv(S), \gamma\rangle]\ Post(S)}{Pre(S), Inv(S) \vdash [\langle R(S', S''), G(S, S'), Inv(S), \mathbf{skip}; \gamma\rangle]\ Post(S)}$$

where G must be reflexive for the rule to hold and the static variable s_0 denotes the state before/after the nonblocking ($\neg\ bv$) stutter step has been executed. The new pre-condition for the rest program γ can be computed as the stable part of $Pre(s_0)$ over the rely condition $R(s_0, S)$. For an empty rest program γ in a sequential composition $\alpha; \gamma$, we simply rewrite α to $(\alpha; \mathbf{last})$ to get the base case.

We execute an assignment $(S := e); \gamma$ according to the following rule

$$\frac{\begin{array}{l} Pre(s_0), Inv(s_0), s_1 = e \vdash G(s_0, s_1) \\ Pre(s_0), Inv(s_0), s_1 = e \vdash Inv(s_1) \\ Pre(s_0), s_1 = e, R(s_1, S), Inv(S), \neg\ bv \vdash [\langle R, G, Inv, \gamma\rangle]\ Post \end{array}}{Pre(S), Inv(S) \vdash [\langle R(S', S''), G(S, S'), Inv(S), (S := e); \gamma\rangle]\ Post(S)}$$

where the static variables s_0/s_1 denote the state vector S before/after the assignment. In the third premise of the rule, the new pre-condition for the rest program γ is typically given by the stable part of $Pre(s_0)$ over the assignment and the subsequent rely $R(s_1, S)$.

The $\mathbf{if}^*$ statement gives two new premises for the *current* state according to the following rule

$$\frac{\begin{array}{c} Pre, Inv, \varphi \vdash [\langle R, G, Inv, \alpha; \gamma\rangle]\ Post \\ Pre, Inv, \neg\ \varphi \vdash [\langle R, G, Inv, \beta; \gamma\rangle]\ Post \end{array}}{Pre, Inv \vdash [\langle R, G, Inv, (\mathbf{if}^* \varphi\ \mathbf{do}\ \alpha\ \mathbf{else}\ \beta); \gamma\rangle]\ Post}$$

The **if** statement introduces a nonblocking stutter step to evaluate its test[7]

$$\frac{Pre, Inv, \varphi \vdash [\langle R, G, Inv, \mathbf{skip}; \alpha; \gamma\rangle]\ Post \quad Pre, Inv, \neg\ \varphi \vdash [\langle R, G, Inv, \mathbf{skip}; \beta; \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, (\mathbf{if}\ \varphi\ \mathbf{do}\ \alpha\, \mathbf{else}\, \beta); \gamma\rangle]\ Post}$$

The $\mathbf{while}^*$ statement gives two new premises for the current state according to the following rule

$$\frac{Pre, Inv, \varphi \vdash [\langle R, G, Inv, \alpha; (\mathbf{while}^*\ \varphi\ \mathbf{do}\ \alpha); \gamma\rangle]\ Post \quad Pre, Inv, \neg\ \varphi \vdash [\langle R, G, Inv, \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, (\mathbf{while}^*\ \varphi\ \mathbf{do}\ \alpha); \gamma\rangle]\ Post}$$

Similarly, the iteration operator * gives two new premises for the current state according to the following rule

$$\frac{Pre, Inv \vdash [\langle R, G, Inv, \alpha; (\alpha^*; \gamma)\rangle]\ Post \quad Pre, Inv \vdash [\langle R, G, Inv, \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, \alpha^*; \gamma\rangle]\ Post}$$

The **while** statement introduces a nonblocking stutter step to evaluate its test according to the following rule

$$\frac{Pre, Inv, \varphi \vdash [\langle R, G, Inv, \mathbf{skip}; \alpha; (\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha); \gamma\rangle]\ Post \quad Pre, Inv, \neg\ \varphi \vdash [\langle R, G, Inv, \mathbf{skip}; \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, (\mathbf{while}\ \varphi\ \mathbf{do}\ \alpha); \gamma\rangle]\ Post}$$

An **await** φ statement followed by a rest program γ executes according to the following rule

$$\frac{Pre, Inv, \varphi \vdash [\langle R, G, Inv, \gamma\rangle]\ Post \quad Pre(s_0), R(s_0, S), Inv(S), \neg\ \varphi(s_0), bv \vdash [\langle R, G, Inv, \mathbf{await}\ \varphi; \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, \mathbf{await}\ \varphi; \gamma\rangle]\ Post}$$

In the first premise, the rest program remains to be executed and φ is added to the current pre-condition. In the second premise, we execute a blocked (bv) stutter step.

The **let** statement rewrites to the following new premise for the current state

$$\frac{Pre, Inv, Y = e, \Box\ Y' = Y'' \vdash [\langle R, G, Inv, \alpha; \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, (\mathbf{let}\ Y = e\ \mathbf{in}\ \alpha); \gamma\rangle]\ Post}$$

where variables Y are fresh (just as variables $\underline{y}$ in Definition (1.9)).

[7] Of course, the implementation in KIV immediately executes the introduced **skip** as explained above.

Similarly, the **choose** statement gives two new premises for the current state

$$\frac{\begin{array}{c} Pre, Inv, \varphi, \Box\ Y' = Y'' \vdash [\langle R, G, Inv, \alpha; \gamma\rangle]\ Post \\ Pre, Inv, \forall\ Y.\ \neg\ \varphi \vdash [\langle R, G, Inv, \beta; \gamma\rangle]\ Post \end{array}}{Pre, Inv \vdash [\langle R, G, Inv, (\textbf{choose}\ Y\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta); \gamma\rangle]\ Post}$$

for fresh variables Y (as variables $\underline{y}$ in Definition (1.10)).

A procedure call PROC is rewritten with its declaration α in the current state

$$\frac{Pre, Inv, Y = S_0, \Box\ Y' = Y'' \vdash [\langle R, G, Inv, \alpha^Y_{S_0}; \gamma\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, \texttt{PROC}(S_0; S); \gamma\rangle]\ Post}$$

using a local copy Y for its input parameters S_0, according to the unfolding axiom (1.11). If PROC is undeclared, then we can *not* execute it. Instead, PROC can be specified by a regular formula φ as $\texttt{PROC} \vdash \varphi$. Then PROC can be rewritten to φ, which can be executed according to Theorem 2.2.

We can also merely *split* a sequential composition into its constituent subprograms with the following RG rule

$$\frac{Pre, Inv \vdash [\langle R, G, Inv, \alpha\rangle]\ Post_\alpha \qquad Post_\alpha, Inv \vdash [\langle R, G, Inv, \beta\rangle]\ Post}{Pre, Inv \vdash [\langle R, G, Inv, \alpha; \beta\rangle]\ Post}$$

where a suitable post-condition $Post_\alpha$ must be given for α.

Weakening/strengthening of rely/guarantee conditions is possible according to the following rule

$$\frac{Pre, Inv \vdash [\langle R_1, G_1, Inv, \alpha\rangle]\ Post_1}{Pre, Inv \vdash [\langle R, G, Inv, \alpha\rangle]\ Post}$$

given that $R \to R_1$, $G_1 \to G$ and $Post_1 \to Post$. (Weakening the pre-condition is possible with basic sequent calculus rules already.)

4.3.3. Induction for RG Assertions

The symbolic execution of RG assertions for partial/total correctness can only consider finite prefixes of an interval. On infinite intervals, the central difference between RG assertions for partial and total correctness results from the fact that the former are safety formulas,[8] but the latter are not, due to their implicit liveness part.

As a consequence, we can only provide a general induction rule to prove RG assertions for partial correctness on infinite intervals (similar to an always formula, see rule (2.5)). To extract a *fresh* counter N for well-founded induction from an RG assertion for partial correctness we first use the equivalence

$$(R \xrightarrow{+} G) \ \leftrightarrow\ \forall\ B.\ \Diamond\ B \to ((R \wedge \neg\ B) \xrightarrow{+} G) \tag{4.3}$$

[8] RG assertions for partial correctness are safety formulas, since formulas $\varphi \xrightarrow{+} \psi$ are safety formulas if $\neg\ \varphi$ and ψ are safety formulas.

where B is a boolean variable, to rewrite a $\xrightarrow{+}$ formula. Then rule (2.4) extracts N from the known liveness property $\diamond\ B$. Together, the following generic induction principle for [.] assertions holds:

$$\frac{N = n, N = N'' + 1\,\mathbf{until}\,B, \mathit{IndHyp}(n), \mathit{Pre}, \mathit{Inv} \vdash [R \wedge \neg\ B, G, \mathit{Inv}, \alpha]\ \mathit{Post}}{\mathit{Pre}, \mathit{Inv} \vdash [R, G, \mathit{Inv}, \alpha]\ \mathit{Post}} \tag{4.4}$$

where the induction hypothesis is

$$\mathbf{A}\ \forall\ S, B, N.\ N \prec n \rightarrow ((N = N'' + 1\,\mathbf{until}\,B) \wedge \mathit{Pre} \wedge \mathit{Inv} \rightarrow [R \wedge \neg\ B, G, \mathit{Inv}, \alpha]\ \mathit{Post})$$

For the verification of RG assertions for *total* correctness on infinite intervals, no general induction principle exists. Hence, an application specific term that decreases during symbolic execution must be found in concrete inductive proofs. In the next example below, variable Y is such a (variant) term; Section 5.2 also uses a variant to prove the total correctness of the parallel FIND algorithm.

4.3.4. Example

Now we illustrate an explicit derivation for the total correctness of a simple sequential program [109] that works on two counters X and Y as follows:

$$\begin{array}{ll} L_1: & \mathbf{while}\ X = 0 \vee Y > 0\ \mathbf{do} \\ L_2: & \quad \mathbf{if}\ X = 0\ \mathbf{then} \\ L_3: & \quad\quad X := 1 \\ L_4: & \quad \mathbf{else}\ Y := Y - 1 \end{array}$$

In the following, $\mathbf{L_i}$ denotes the program above from line L_i onwards (possibly followed by the loop again) where $i \in \{1..4\}$. As rely condition we use here

$$R \equiv X' = X'' \wedge (X' \neq 0 \rightarrow Y' = Y'')$$

which states that X is not changed concurrently and Y remains unchanged by others only if X is not 0 before the environment transition. Otherwise, the environment can change the value of Y nondeterministically. As a pre-condition we use $X = 0$ and all other RG conditions are trivial.

The proof of total correctness for the example program corresponds to the following derivation tree (read upwards starting at the bottom of the proof tree)

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{X = 1, Y + 1 = n, \mathit{IndHyp}(n) \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_1}\rangle\ \mathit{true}}{X = 1, Y = n > 0, \mathit{IndHyp}(n) \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_4}\rangle\ \mathit{true}}\ \text{assign}}{X = 1, Y = n > 0, \mathit{IndHyp}(n) \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_2}\rangle\ \mathit{true}}\ \text{if, skip}}{X = 1, Y = n, \mathit{IndHyp}(n) \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_1}\rangle\ \mathit{true}}\ \text{while, skip}}{X = 1 \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_1}\rangle\ \mathit{true}}\ \text{induction (2.3) with } e \equiv Y$$

$$\cfrac{\cfrac{\cfrac{\cfrac{X = 0 \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_3}\rangle\ \mathit{true}}{X = 0 \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_2}\rangle\ \mathit{true}}\ \text{assign}}{X = 0 \vdash \langle R, \mathit{true}, \mathit{true}, \mathbf{L_1}\rangle\ \mathit{true}}\ \text{if, skip}}$$

and the resulting premise (at the top of the derivation tree) is closed by the induction hypothesis

$$IndHyp(n) \equiv \mathbf{A}\ \forall\ X, Y.\ Y < n \rightarrow (X = 1 \rightarrow \langle R, true, true, \mathbf{L_1} \rangle\ true)$$

instantiating X/Y with X/Y, respectively.

RG proofs in KIV for sequential programs execute the introduced rules mainly automatically. However, to deal with loops, induction rules are typically applied manually. This interactive step often also requires to introduce loop invariants as explained in Example 2.3.

5. RG Reasoning for Two-Interleaving

The symbolic execution of *interleaved* programs is not practicable even for simple programs due to the well-known state explosion problem. Therefore, decomposition rules that reduce the verification of a parallel system to the verification of its sequential subprograms are desirable. In this chapter, we introduce such decomposition rules for programs with two interleaved components.

The structure of this chapter is as follows: Section 5.1 introduces basic RG decomposition rules for partial/total correctness of a system with two interleaved components and proves their soundness. Then we extend these basic rules with additional predicates that are typically required in applications and illustrate the use of one such rule to compositionally verify total correctness of the parallel FIND algorithm, in Section 5.2. Finally, Section 5.3 gives an extension of our RG rules for the compositional verification of absence of deadlock and illustrates its use to verify a simple mutex algorithm correct. We have mechanized all proofs in KIV. The exposition in Section 5.1 follows our article [89], the rest has not been described before.

5.1. Basic RG Rule

To start with, we consider a basic RG rule for a program that interleaves two undeclared procedures $\texttt{PROC}_1$ and $\texttt{PROC}_2$. The rule is "basic" as it considers neither pre-/post-conditions, invariants or total correctness. This helps us to focus on the central aspects of the soundness proof that applies the basic deduction principles from the first part of this work.

Theorem 5.1 (Basic RG Rule for Two-Interleaving)
The following basic RG decomposition rule is correct:

$$\frac{\begin{array}{l}(1) \vdash G_1(S,S)\\ (2) \vdash R_1(S,S_0) \wedge R_1(S_0,S_1) \rightarrow R_1(S,S_1)\\ (3) \vdash G_1(S,S_0) \rightarrow R_2(S,S_0)\\ (4) \vdash [R_1, G_1, \mathit{true}, \texttt{PROC}_1(S)]\ \mathit{true}\\ \\ (5) \vdash G_2(S,S)\\ (6) \vdash R_2(S,S_0) \wedge R_2(S_0,S_1) \rightarrow R_2(S,S_1)\\ (7) \vdash G_2(S,S_0) \rightarrow R_1(S,S_0)\\ (8) \vdash [R_2, G_2, \mathit{true}, \texttt{PROC}_2(S)]\ \mathit{true}\end{array}}{\mathit{true} \vdash [R_1 \wedge R_2, G_1 \vee G_2, \mathit{true}, \texttt{PROC}_1(S) \parallel_{\mathsf{nf}} \texttt{PROC}_2(S)]\ \mathit{true}}$$

Conclusion of the Basic Rule. The conclusion of the rule states that each transition of the interleaved system preserves either G_1 or G_2 as long as the previous environment transitions satisfy both R_1 and R_2. Since a global system step is taken by either process 1 or 2, the global guarantee is the disjunction of the local guarantee conditions G_i for $i = 1, 2$. Similarly, since a local environment transition of an individual process i can contain transitions of the global environment, these must preserve each individual rely condition R_i.

Premises of the Basic Rule. The rule has predicate logic premises (1) to (3) for process 1 and (5) to (7) for process 2.[1] They state that the individual guarantees are reflexive (premises (1) and (5)) and thus stutter transitions that leave the state unchanged are admissible. To group several subsequent transitions of other processes into one rely-transition, the rely conditions must be transitive (premises (2) and (6)). The two central RG restrictions (3) and (7) have already been explained. They are often ensured *canonically* by choosing the guarantee of one process as the rely of all other processes. (In Section 6.1 we also use an instance of an individual guarantee that is strictly stronger than just the relies of all other processes.) The two remaining temporal logic premises (4) and (8) require to prove an RG assertion for partial correctness for each individual process.

Soundness Proof of RG Rule 5.1. In the remainder of this section, we illustrate the derivation of rule (5.1) in RGITL. The two central deduction principles that we use are the symbolic execution of the interleaving operator (see Theorem 2.2 and Appendix A.1) and the compositionality properties of the chop, the star and the interleaving operator (see compositionality rule (1.1)).

The correctness proof of the basic rule reduces to showing that

$$[R_1 \xrightarrow{+} G_1] \parallel_{\mathsf{nf}} [R_2 \xrightarrow{+} G_2] \vdash R \xrightarrow{+} G \tag{5.1}$$

can be derived from its premises where $R \equiv R_1 \wedge R_2$ and $G \equiv G_1 \vee G_2$: The correctness of the rule then simply follows with the definition of the [.] operator which expands the RG assertions for partial correctness to $\xrightarrow{+}$ formulas and by application of the compositionality rule (1.1) to substitute each procedure with its corresponding sustains formula according to premises (4) and (8) above. (Technically, the goal above is the last premise of the compositionality rule (1.1) while the first two premises follow from premises (4) and (8) above.)

Thus it remains to prove (5.1). The proof is by induction over $R \xrightarrow{+} G$, using (4.3) and (2.4) to retrieve a counter for well-founded induction. The resulting additional formulas and the higher-order variable *IH* for lazy induction are omitted in the following for brevity. Then the proof applies symbolic execution according to Theorem 2.2, based on the unwinding rule (4.2) for $\xrightarrow{+}$ formulas.

Thus a symbolic execution step of the *succedent* formula in (5.1) requires to prove G_1 or G_2 for the first system transition. Then the leading operator $\bullet$ is removed, giving the original succedent again (together with the additional assumption that the first global

[1] In the following, we typically ignore the restriction that predicate logic axioms must not use flexible variables. This reduces extra syntax for static variables with indices.

environment transition has satisfied R). At the same time, symbolic execution of the interleaving in the *antecedent* of (5.1) gives 6 cases, depending on whether the scheduled component terminates, executes an unblocked transition or a blocked transition (see Definition 1.4 and Appendix A.1, respectively). We consider w.l.o.g. the first three, where the first component is executed.

Case 1: Component 1 Terminates. In this case the goal reduces to

$$R_2 \xrightarrow{+} G_2 \vdash R \xrightarrow{+} G$$

which is simple to prove with one further symbolic execution step after which the goal is repeated and closed by applying the induction hypothesis.[2]

Case 2: Component 1 Executes an Unblocked Step. This gives two sub-cases: The simple case where the first local environment transition of component 1 satisfies R_1 leads to a goal that repeats (5.1) and can thus be closed with the induction hypothesis. (Observe that property $R_1 \xrightarrow{+} G_1$ of component 1 ensures G_1 for its program transition, which becomes globally visible.)

The difficult second case where the first local environment transition of component 1 violates R_1 must be lead to a contradiction. A symbolic execution step in this case leads to

$$G_1(s_0, s_1),\ \neg\, R_1(s_1, u),\ R_1(s_1, S), (u = S) \parallel_{\mathsf{nf}} (R_2 \xrightarrow{+} G_2) \vdash R \xrightarrow{+} G \qquad (5.2)$$

where s_0/s_1 is the state before/after the system transition of the first component. Formula $R_1(s_1, S)$ is implied by the global rely R from unwinding the $\xrightarrow{+}$ formula in the succedent, and formula $\neg\, R_1(s_1, u)$ characterizes the first *local* environment transition of component 1. Hence, state u (introduced by existential quantification as described in Appendix A.1) is the state after the first local environment transition of component 1. Only when component 1 is executed again, state u becomes the next state of the global run ($u = S$) which immediately closes the proof by the contradiction between $\neg\, R_1(s_1, u)$ and $R_1(s_1, S)$. In this case, the first local environment transition of component 1 is the first global environment transition.

Otherwise, when the next program transition is by component 2, it must be integrated into the local environment transition of component 1 (as explained in Section 1.4/Appendix A.1) and further symbolic execution is necessary to conclude: The second symbolic execution step by component 2 introduces new static variables s_2, s_3 for S and S' which satisfy $G_2(s_2, s_3)$ according to executing $\xrightarrow{+}$ for component 2. Assumption $R_1(s_1, S)$ thus becomes $R_1(s_1, s_2)$, and $R_1(s_3, S)$ holds again from unwinding $\xrightarrow{+}$ in the succedent. Together, formulas

$$R_1(s_1, s_2) \wedge G_2(s_2, s_3) \wedge R_1(s_3, S)$$

[2] Lazy induction saves us from defining a new induction hypothesis to prove the sequent above. The goal could also be closed with a trivial rule that shows that weakening/strengthening of RG conditions in $\xrightarrow{+}$ formulas is admissible.

imply that from the local view of component 1, a sequence of an overall environment transition followed by a transition of component 2 and then again an overall environment transition satisfies the local rely $R_1(s_1, S)$ from s_1 to S according to the given predicate logic side conditions $G_2 \rightarrow R_1$ and transitivity of R_1. This is the core argument of the soundness proof: Sequences of global rely transitions and guarantee transitions of the other component together form a valid local rely transition R_1 for component 1.

The second symbolic execution step above also introduces a new static variable v for the state of component 2 after its local environment transition. Symmetrically, we must now discern whether the local rely $R_2(s_3, v)$ holds. If it holds, then goal (5.2) is repeated and the proof obligation can be closed with the induction hypothesis. Otherwise, the second symbolic execution step yields the following sequent:

$$\begin{aligned} &\neg\, R_1(s_1, u), R_1(s_1, S), G_2(s_2, s_3), \neg\, R_2(s_3, v), R_2(s_3, S), [u = S] \parallel_{\mathsf{nf}} [v = S] \quad (5.3) \\ &\vdash R \xrightarrow{+} G \end{aligned}$$

But now a further symbolic execution step closes the goal regardless whether component 1 or 2 is executed, since both $u = S$ and $v = S$ give the required contradiction in the antecedent.

Case 3: Component 1 Executes a Blocked Step. In this case, component 2 executes a step too, and the proof is largely symmetric. (In particular, when both relies are false immediately, the resulting goal corresponds to (5.3).) This concludes the derivation of rule 5.1 in RGITL using symbolic execution.

Variants of the Basic Rule. Of course, the basic RG rule 5.1 also holds for weak-fair interleaving $\parallel$ since weakly-fair interleaved intervals are a subset of unfairly interleaved intervals. We have also derived the correctness of the rule for the total correctness operator $\langle\ .\ \rangle$, which essentially requires to additionally prove

$$((R_1 \xrightarrow{+} G_1) \wedge (\Box\ R_1 \rightarrow \Diamond\ \mathbf{last})) \parallel_{\mathsf{nf}} ((R_2 \xrightarrow{+} G_2) \wedge (\Box\ R_2 \rightarrow \Diamond\ \mathbf{last})), \Box\ R \vdash \Diamond\ \mathbf{last}$$

according to Definition 4.2, but ignoring invariant propagation for brevity.

To prove the sequent above, we first rewrite the implication in component 1 as disjunction which leads w.l.o.g. to the following subgoal

$$((R_1 \xrightarrow{+} G_1) \wedge \Diamond \neg\, R_1) \parallel_{\mathsf{nf}} ((R_2 \xrightarrow{+} G_2) \wedge (\Box\ R_2 \rightarrow \Diamond\ \mathbf{last})), \Box\ R \vdash \Diamond\ \mathbf{last}$$

where fairness rule (2.8) can be applied to get a counter N for induction. Then we additionally apply fairness rule (2.7) to deal with the case where component 1 is preempted and not resumed again, before N decrements. The remaining proof applies similar arguments as we have already discussed in the proof of rule 5.1.

5.2. Extended RG Rule

This section introduces an extended RG rule for the decomposition of partial/total correctness assertions of a system with 2 interleaved components. Different from the basic rule 5.1 it additionally accounts for pre-/post-conditions and an invariant predicate. These additional predicates are typically used in concrete applications and we show the instantiation of the extended rule to compositionally verify the total correctness of the parallel FIND algorithm from Figure 1.1.

5.2.1. Extended RG Rule for Two-Interleaving

Theorem 5.2 (Extended RG Rule for Two Interleaved Components)
The following extended RG rule for partial/total correctness of an interleaved program with two components is correct.

$$
\begin{array}{l}
(1) \vdash G_1(S,S) \\
(2) \vdash R_1(S,S_0) \wedge R_1(S_0,S_1) \rightarrow R_1(S,S_1) \\
(3) \vdash G_1(S,S_0) \rightarrow R_2(S,S_0) \\
(4) \vdash \mathit{Pre}_1(s,S_0) \wedge R_1(S_0,S_1) \rightarrow \mathit{Pre}_1(s,S_1) \\
(5) \vdash \mathit{Post}_1(s,S_0) \wedge R_1(S_0,S_1) \rightarrow \mathit{Post}_1(s,S_1) \\
(6)\ \mathit{Pre}_1(s,S), \mathit{Inv}(S) \vdash [\langle R_1, G_1, \mathit{Inv}, \texttt{PROC}_1(S)\rangle]\ \mathit{Post}_1(s,S)
\end{array}
$$

$$
\begin{array}{l}
(7) \vdash G_2(S,S) \\
(8) \vdash R_2(S,S_0) \wedge R_2(S_0,S_1) \rightarrow R_2(S,S_1) \\
(9) \vdash G_2(S,S_0) \rightarrow R_1(S,S_0) \\
(10) \vdash \mathit{Pre}_2(s,S_0) \wedge R_2(S_0,S_1) \rightarrow \mathit{Pre}_2(s,S_1) \\
(11) \vdash \mathit{Post}_2(s,S_0) \wedge R_2(S_0,S_1) \rightarrow \mathit{Post}_2(s,S_1) \\
(12)\ \mathit{Pre}_2(s,S), \mathit{Inv}(S) \vdash [\langle R_2, G_2, \mathit{Inv}, \texttt{PROC}_2(S)\rangle]\ \mathit{Post}_2(s,S)
\end{array}
$$

$$
\frac{(13) \vdash (\exists\ S.\ \mathit{Init}(S)) \wedge (\mathit{Init}(S) \rightarrow \mathit{Pre}_1(S,S) \wedge \mathit{Pre}_2(S,S) \wedge \mathit{Inv}(S))}{\begin{array}{r}\mathit{Init}(S), s = S \vdash [\langle R_1 \wedge R_2, G_1 \vee G_2, \mathit{Inv}, \texttt{PROC}_1(S) \parallel_{\mathsf{nf}} \texttt{PROC}_2(S)\rangle] \\ (\mathit{Inv}(S) \wedge \mathit{Post}_1(s,S) \wedge \mathit{Post}_2(s,S))\end{array}}
$$

Compared to the basic rule 5.1, this rule adds the predicate logic restrictions that the pre- and post-conditions are stable with respect to the rely (premises (4), (5) for component 1 and (10), (11) for component 2). This is necessary, because a process might start after/terminate before the other one. Moreover, there must exist an initial overall system state that satisfies predicate *Init* and establishes the two preconditions Pre_1 and Pre_2 for both processes and the invariant according to premise (13).

The soundness proof of rule 5.2 applies similar arguments as already discussed in the proof of the basic rule 5.1. Of course, the rule also holds for weakly-fair interleaved programs.

$$
\begin{array}{l}
\mathrm{FIND}(Ar, Out)\ \{ \\
\quad \textbf{let}\ Out_E = \#Ar, Out_O = \#Ar\ \textbf{in}\ \{ \\
\qquad \mathrm{FIND}_E(Out_O, Ar, Out_E) \parallel_{\mathsf{nf}} \mathrm{FIND}_O(Out_E, Ar, Out_O); \\
\qquad Out := min(Out_E, Out_O) \\
\quad \}\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{FIND}_E(Out_O, Ar, Out_E)\ \{ \\
\quad \textbf{let}\ Done = false, Ix = 0\ \textbf{in}\ \{ \\
\qquad \textbf{while}\ \neg\ Done \wedge Ix < Out_O\ \textbf{do}\ \{ \\
\qquad\quad \textbf{if}\ pred(Ar[Ix])\ \textbf{then} \\
\qquad\qquad Out_E := Ix, Done := true \\
\qquad\quad \textbf{else}\ Ix := Ix + 2 \\
\qquad \}\ \}\ \}
\end{array}
\qquad
\begin{array}{l}
\mathrm{FIND}_O(Out_E, Ar, Out_O)\ \{ \\
\quad \textbf{let}\ Done = false, Ix = 1\ \textbf{in} \\
\qquad \textbf{while}\ \neg\ Done \wedge Ix < Out_E\ \textbf{do}\ \{ \\
\qquad\quad \textbf{if}\ pred(Ar[Ix])\ \textbf{then} \\
\qquad\qquad Out_O := Ix, Done := true \\
\qquad\quad \textbf{else}\ Ix := Ix + 2 \\
\qquad \}\ \}\ \}
\end{array}
$$

Figure 5.1.: Specification of FIND in RGITL.

5.2.2. Compositional Verification of the Parallel FIND Algorithm

In this section, we illustrate the application of rule 5.2 to compositionally verify the total correctness of the parallel FIND algorithm from the introduction, Figure 1.1.

Specifying FIND in RGITL. We start by specifying the algorithms in the abstract programming language of RGITL. Figure 5.1 shows the result: We use an unspecified predicate $pred\colon elem \to bool$ over elements of a generic type $elem$ and an algebraic array Ar of these elements. As soon as a process finds a position where $pred$ holds it notifies the other process by setting variable Out_E/Out_O to its current position and terminates. The result of the parallel search is the minimum of both individual results that is finally stored in Out. (Remember that the programming language does not have a return statement, so a return is modeled by assigning the result to Out at the end of an operation.)

The Total Correctness of FIND. The overall total correctness theorem that we want to prove for FIND is:

$$
\begin{array}{l}
\quad Ar = ar \\
\vdash \langle Ar' = Ar'' \wedge Out' = Out'', true, true, \mathrm{FIND}(Ar, Out)\rangle\ Out = minpos(ar)
\end{array}
\tag{5.4}
$$

That is, if FIND runs in a global environment that does not change the array and the output, then it terminates and returns the minimal array position where predicate $pred$ holds, or the length of the array if $pred$ does not hold at any position. Function $minpos$ computes this minimal position; it is algebraically specified with the following

two simple axioms:

$$\begin{array}{l} \quad (\exists\ n.\ n < \#Ar \wedge pred(Ar[n])) \\ \rightarrow \quad minpos(Ar) < \#Ar \wedge pred(Ar[minpos(Ar)]) \\ \quad \wedge\ \forall\ m.\ m < minpos(Ar) \rightarrow \neg\ pred(Ar[m]) \end{array}$$

$$(\forall\ n.\ n < \#Ar \rightarrow \neg\ pred(Ar[n])) \rightarrow minpos(Ar) = \#Ar$$

The proof of (5.4) first unfolds the and then the **let** to introduce variables Out_E/Out_O. Then it applies the split rule for the sequential composition operator (see Section 4.3) where program α becomes $\text{FIND}_E \parallel_{\mathsf{nf}} \text{FIND}_O$, and β is the assignment $Out := minpos(Out_E, Out_O)$. This gives us two remaining proof obligations according to the two premises of the splitting rule where the intermediate assertion $Post_\alpha$ from the rule is simply $Post_E \wedge Post_O$ as given below. The second premise then follows trivially with our instantiation of the RG conditions below and the RG rule for assignments.[3]

It remains to prove the following RG assertion for total correctness of the interleaved system $\text{FIND}_E \parallel_{\mathsf{nf}} \text{FIND}_O$

$$\begin{array}{l} \quad ar = Ar, Out_E = \#Ar, Out_O = \#Ar \\ \vdash \langle S' = S'', G_E(S,S') \vee G_O(S,S'), Inv(S), \text{FIND}_E(S) \parallel \text{FIND}_O(S) \rangle \\ \quad (Inv(S) \wedge Post_E(ar,S) \wedge Post_O(ar,S)) \end{array}$$

where state parameter S is the vector Out_E, Out_O, Ar. Here we assume that the overall environment never changes S which follows immediately from our overall environment assumption $Ar' = Ar''$ in (5.4) and the locality $\Box\ (Out_E{}' = Out_E{}'' \wedge Out_O{}' = Out_O{}'')$ which we discard after merging it into the RG assertion. (The empty global environment also ensures the individual rely conditions R_E/R_O given below.)

Next the proof applies the extended RG rule 5.2. The instances for its RG predicates are as follows:

RG Instances for FIND. The global initialization predicate *Init* is just

$$Init(S) \equiv Out_E = Out_O = \#Ar$$

The pre-conditions for each process ensure that the results Out_E/Out_O are equal to the length of the current array which must be still equal to the array in the initial overall system state ($ar = Ar$).

$$\begin{array}{l} Pre_E(ar,S) \equiv Out_E = \#Ar \wedge ar = Ar \\ Pre_O(ar,S) \equiv Out_O = \#Ar \wedge ar = Ar \end{array}$$

[3]Note that formulas such as the locality property $\Box\ Out_E{}' = Out_E{}''$ that hold for the full intervals of FIND, also hold in each of its subintervals. Hence, we can use the fact that the global environment never changes Out_E/Out_O both for the verification of $\text{FIND}_E \parallel_{\mathsf{nf}} \text{FIND}_O$ and the assignment $Out := minpos(Out_E, Out_O)$.

The rely conditions state that the array and the result variables are not concurrently changed.

$$R_E(S', S'') \equiv$$
$$Ar' = Ar'' \wedge Out_E{}' = Out_E{}'' \wedge Out_O{}'' \leq Out_O{}' \wedge (Out_O{}' < \#Ar' \rightarrow Out_O{}' = Out_O{}'')$$
$$R_O(S', S'') \equiv$$
$$Ar' = Ar'' \wedge Out_O{}' = Out_O{}'' \wedge Out_E{}'' \leq Out_E{}' \wedge (Out_E{}' < \#Ar' \rightarrow Out_E{}' = Out_E{}'')$$

Moreover, the other process never increments its result and once it is set to a valid position (i.e., one which is less than the length of Ar'), it remains unchanged.

The guarantees are defined canonically as the rely conditions of the other process:

$$G_E \equiv R_O \quad \text{and} \quad G_O \equiv R_E$$

The invariant predicate Inv states that the shared result variables never exceed the size of the array.

$$Inv(S) \equiv$$
$$Out_E \leq \#Ar \wedge Out_O \leq \#Ar$$
$$\wedge\ (Out_E < \#Ar \rightarrow \mathit{fbelow}(Out_E, 0, Ar)) \wedge (Out_O < \#Ar \rightarrow \mathit{fbelow}(Out_O, 1, Ar))$$

Moreover, once the shared result is set to a valid position, predicate $pred$ evaluates to false for all even/odd indices below Out_E/Out_O according to formulas $\mathit{fbelow}(Out_E, 0, Ar)$ and $\mathit{fbelow}(Out_O, 1, Ar)$ where

$$\mathit{fbelow}(m, n, Ar) \equiv \forall\ k.\ k < m \wedge k\ mod\ 2 = n \rightarrow \neg\ pred(Ar[k])$$

The post-conditions are slightly more difficult since they must consider the three possible outcomes of an individual search: Either a valid position is found and $Out_E/Out_O < \#Ar$, or the search has been interrupted by the other process that has found a valid position below the current index, or no position has been found:

$$Post_E(ar, S) \equiv ar = Ar \wedge$$
$$(\quad Out_E < \#Ar \wedge even(Out_E) \wedge pred(Ar[Out_E]) \wedge \mathit{fbelow}(Out_E, 0, Ar)$$
$$\vee\ (Out_E = \#Ar \wedge Out_O < Out_E \wedge \forall\ k.\ k < Out_O \rightarrow \neg\ pred(Ar[k]))$$
$$\vee\ Out_E = \#Ar \wedge \mathit{fbelow}(Out_E, 0, Ar))$$
$$Post_O(ar, S) \equiv ar = Ar \wedge$$
$$(\quad Out_O < \#Ar \wedge odd(Out_O) \wedge pred(Ar[Out_O]) \wedge \mathit{fbelow}(Out_O, 1, Ar)$$
$$\vee\ (Out_O = \#Ar \wedge Out_E < Out_O \wedge \forall\ k.\ k < Out_E \rightarrow \neg\ pred(Ar[k]))$$
$$\vee\ Out_O = \#Ar \wedge \mathit{fbelow}(Out_O, 1, Ar))$$

Proving an RG Assertion for FIND$_E$. With the instances given above, the predicate logic premises of rule 5.2 are trivial. It remains to prove the central RG assertion for total correctness of operation FIND$_E$ according to premise (6) of rule 5.2.

(The proof of the RG assertion for FIND_O is symmetric and therefore not described here.)

$$Pre_E(ar, S), Inv(S) \vdash \langle R_E, G_E, Inv(S), \text{FIND}_E(S)\rangle \; Post_E(ar, S)$$

The proof starts by unfolding the call and the **let** (see Section 4.3) and generalizing the additional precondition $Ix = 0$ to the following loop invariant

$$fbelow(Ix, 0, Ar) \wedge even(Ix)$$

Then the proof continues by applying induction rule (2.3) using the variant $\#Ar - Ix$ and symbolic execution for sequential programs according to Section 4.3:

Symbolically executing the while loop gives two cases. In the first case the loop exits immediately and the post-condition has to be shown (this is trivial by expanding the definition). The second case steps through the loop. Executing an assignment creates two side goals where the guarantee and the preservation of the invariant have to be proved (again these predicate logic goals follow immediately from the definitions). Symbolically executing the if-conditional gives two cases (evaluating the test takes a step; the trivial sidegoals for a skip step are closed immediately): In the first case, the loop is exited since the element has been found, and the post-condition has to be proved again by unfolding the definition. The other case, where the loop has to be repeated is closed with the induction hypothesis.

5.3. Absence of Deadlock

The verification of parallel algorithms *with locks* typically includes showing the *absence of deadlock* for the overall concurrent system. This section introduces our approach for the compositionally verifying absence of deadlock ($\Box \neg$ **blocked**) and illustrates it using a simple mutex algorithm. The basic idea is adapted from [109] to our formalism and we have mechanically verified the soundness of the decomposition in KIV.

Following [109], we extend the set of RG conditions with a predicate $Runs\colon state \to bool$ that defines those states where a program `PROC` is not blocked. An RG assertion for partial correctness *and* absence of deadlock uses predicate *Runs* as follows:[4]

Definition 5.1 (Partial Correctness and Absence of Deadlock)

$$Pre \vdash [R, G \wedge (Inv \wedge Runs \to \neg \textbf{ blocked}), Inv, \texttt{PROC}] \; Post$$

denotes an RG assertion for partial correctness and absence of deadlock.

The additional guarantee $Inv \wedge Runs \to \neg$ **blocked** ensures that each invariant state that satisfies predicate *Runs* must not be blocked.

[4] Total correctness implies absence of deadlock assuming that the overall system's environment does not provide a deadlock prevention mechanism.

5.3.1. A RG Rule for Absence of Deadlock for Two-Interleaving

Our RG rule that additionally ensures absence of deadlock of a parallel system with two components is just an instance of rule 5.2 for partial correctness with the following adaptations to exclude deadlocks:

In the crucial RG assertions (6) and (12) of rule 5.2, the guarantee conditions are strengthened according to Definition 5.1 using two additional predicates $Runs_1$ for component 1 in premise (6) and $Runs_2$ for component 2 in premise (12). These predicates must adhere to the following additional predicate logic side conditions:

$$(14) \vdash Inv(S) \rightarrow Runs_1(S) \vee Runs_2(S)$$
$$(15) \vdash (Post_1(s,S) \rightarrow Runs_2(S)) \wedge (Post_2(s,S) \rightarrow Runs_1(S))$$

That is, in each state where the invariant holds at least one component must not be blocked and when the post-condition of one component holds, the other component must be in a running state.

With the instances and restrictions above, the conclusion of rule 5.2 ensures that no step of the overall parallel system is ever blocked:

$$\begin{array}{l} \quad Init(S), s = S \\ \vdash [\langle R_1 \wedge R_2, (G_1 \vee G_2) \wedge \neg\, \mathbf{blocked}, Inv, \mathtt{PROC}_1(S) \parallel_{\mathsf{nf}} \mathtt{PROC}_2(S)\rangle] \\ \quad (Inv(S) \wedge Post_1(s,S) \wedge Post_2(s,S)) \end{array}$$

Note that the rule ensures absence of deadlock even for unfair interleaving. This is because our unfair interleaving operator disallows executions where a blocked process infinitely often executes blocked transitions, while the other process that holds a lock is never executed again (which would result in a deadlock). Instead, Case 3 of Definition 1.4 enforces that when a process is blocked, the other process is chosen for execution.[5] This scheduling behavior is in accordance with typical preemptive process-scheduling in operating systems.

5.3.2. Compositional Verification of a Simple Mutex

In this section, we illustrate the compositional proof of partial correctness and absence of deadlock of a simple mutex algorithm by applying our compositional RG rule for absence of deadlock.

Figure 5.2 shows the specification of the algorithm in RGITL. Synchronization is based on a shared counter N that is initially 1 and updated to 0 whenever a process succesfully acquires the lock for N, causing the other process to wait until N is reset to 1 again. The boolean labels L_1 for process 1 and L_2 for process 2 are auxiliary variables that we use to characterize the critical sections of each process.

The overall correctness theorem that we want to prove for `MUTEX` is:

$$Init(S) \vdash [S' = S'', (G_1 \vee G_2) \wedge \neg\, \mathbf{blocked}, Inv(S), \mathtt{MUTEX}(S)]\ true \tag{5.5}$$

[5] Our unfair scheduling semantics can only ignore a component that never blocks according to fairness rule (2.7).

```
MUTEX(L1, L2, N) {
  MUTEX1(L1, N) ||nf MUTEX2(L2, N)
}

MUTEX1(L1, N) {
  while true do {
    await N > 0;            // * Acquire lock N *
    N := 0, L1 := true;     // * Enter critical section *
    skip;                   // * In critical section *
    N := 1, L1 := false;    // * Release lock N *
  } }

MUTEX2(L2, N) {
  while true do {
    await N > 0;            // * Acquire lock N *
    N := 0, L2 := true;     // * Enter critical section *
    skip;                   // * In critical section *
    N := 1, L2 := false;    // * Release lock N *
  } }
```

Figure 5.2.: Specification of a Simple Mutex in RGITL.

where S is the vector of variables L_1, L_2, N. The overall post-condition is trivial here, since the system never terminates. The proof of (5.5) applies the previously introduced RG rule. Suitable instances for its RG predicates are given in the following.

RG Instances for MUTEX. The initialization of the overall system state is $L_1 = L_2 = \mathit{false}$ and $N = 1$ according to predicate *Init*. The pre-conditions of processes 1/2 simply require that the labels L_1/L_2 are false, i.e., each process asserts that it is not in its critical section at the beginning of its operation, i.e.,

$$\mathit{Pre}_1(L_1) \equiv \neg\, L_1 \quad \text{and} \quad \mathit{Pre}_2(L_2) \equiv \neg\, L_2$$

The rely conditions of processes 1/2 merely state that the labels L_1/L_2 are not concurrently changed. The guarantees of processes 1/2 are set canonically as the rely condition of the other process, respectively.

The invariant states the mutual exclusion property, i.e., both process are never in the critical section at same time $\neg\, L_1 \vee \neg\, L_2$. Furthermore, it relates the labels of each process to the value of the lock in the obvious way:

$$\mathit{Inv}(S) \equiv (\neg\, L_1 \vee \neg\, L_2) \wedge (L_1 \vee L_2 \rightarrow N = 0) \wedge (\neg\, L_1 \wedge \neg\, L_2 \rightarrow N = 1)$$

The two individual *Runs* predicates simply require that the *other* process is not in its critical section, because then the *current* process is not forced to wait for a lock.

$$\mathit{Runs}_1(S) \equiv \neg\, L_2 \quad \text{and} \quad \mathit{Runs}_2(S) \equiv \neg\, L_1$$

The post-conditions for process 1/2 are canonically defined as $Runs_2/Runs_1$ to satisfy the predicate logic side condition (15) of the RG rule above.

Proof Sketch for $\texttt{MUTEX}_1$. With these instances, the predicate logic premises of the RG rule, including (14) and (15), are easy to prove. It remains to prove the central RG assertion for operation $\texttt{MUTEX}_1$ according to premise (6). (The proof of the RG assertion for $\texttt{MUTEX}_2$ is symmetric.)

$$\begin{aligned} & Pre_1(L_1), Inv(L_1, L_2, N) \\ \vdash\ & [R_1, G_1 \wedge (Inv \wedge Runs_1 \rightarrow \neg\ \mathbf{blocked}), Inv, \texttt{MUTEX}_1]\ Post_1(L_1) \end{aligned}$$

The proof starts by unfolding the call and applying induction rule (4.4) for the RG assertion in the succedent. The rest of the proof is by symbolic execution of sequential programs as described in Section 4.3:

Symbolically executing the first step enters the while loop, the guarantee and the invariant hold trivially for this stutter step. The second symbolic execution step of the await statement gives two cases:

i) If the lock can not be acquired now, i.e., $N = 0$, then the program executes a blocked stutter step and we have to show that this step sustains the additional guarantee $Inv \wedge Runs_1 \rightarrow \neg\ \mathbf{blocked}$. That is, we must show that it is impossible that predicate $Runs_1$ holds in the current state. Since L_1 is *false* and N is 0 now, the third conjunct of the invariant Inv implies that L_2 must be *true*, i.e., the other process is currently in its critical section. Hence, predicate $Runs_1$ is false by definition. The remaining sequent just repeats the previous one and is discarded by applying the induction hypothesis.

ii) If the lock is free in the current state, i.e., $N = 1$, then the proof continues with two further symbolic execution steps. Proving that these steps sustain the guarantee and invariant just unfolds the definitions. The remaining case where the loop has to be repeated is closed with a suitable induction hypothesis. This concludes the proof.

Note that we are using lazy induction here, as explained in Section 2.2. Hence we only have to define a higher-order variable *IH* for induction once and can apply suitable induction hypotheses for both the case in which the algorithm waits in front of the critical section and also for the case in which it reiterates the loop. (Of course, we could have used two explicit induction hypothesis instead.)

6. RG Reasoning for n-Interleaving

In this chapter, we introduce a RG rule for the compositional verification of programs that interleave an arbitrary finite number of components. We illustrate its use to compositionally verify a concurrent version of the Sieve of Eratosthenes [54] in Section 6.1. Moreover, we derive a state-local rule in Section 6.2, which simplifies the specification/verification based on an overall program state that contains an arbitrary number of local states, to just two representative local states. This can be particularly useful when verifying concurrent data-structures due to the common symmetry of the underlying algorithms. (We show a challenging application of our state-local rule in Section 12.2, in the context of non-blocking garbage collection for lock-free data structures.) The basic ideas of Section 6.1 can also be found in our articles, e.g., [7, 89]. The ideas of state-local reasoning have been introduced in our article [96].

6.1. Generic RG Rule

This section introduces a RG decomposition rule for partial/total correctness of a program that interleaves an arbitrary finite number of components. The rule generalizes our previous rule 5.2 for two interleaved components to the general setting with $n+1$ components, as we explain in the following.

To begin with, we introduce our concurrent system model $\texttt{SPAWN}_n$[1]

$$
\begin{array}{l}
\texttt{SPAWN}_n(S)\ \{ \\
\quad \textbf{if*}\ n = 0\ \textbf{then} \\
\quad\quad \texttt{PROC}_0(S) \\
\quad \textbf{else} \\
\quad\quad \texttt{PROC}_n(S)\ \|_{\mathsf{nf}}\ \texttt{SPAWN}_{n-1}(S) \\
\}
\end{array}
$$

that interleaves $n+1$ processes ($n\colon nat$) where each process $p \leq n$ executes an undeclared procedure $\texttt{PROC}_p$ on the overall system state S.

Next, we generalize the overall system's RG conditions from rule 5.2. Remember, the overall system rely condition for two processes is $R_1 \wedge R_2$, the guarantee $G_1 \vee G_2$ and the pre- and post-conditions are $Pre_1 \wedge Pre_2$ and $Post_1 \wedge Post_2$. In our general setting with $n+1$ components we thus get:

$$
\begin{array}{l}
R{\leq_n}(S', S'') \ \equiv\ \forall\, p \leq n.\ R_p(S', S'') \quad \text{and} \quad G{\leq_n}(S, S') \ \equiv\ \exists\, p \leq n.\ G_p(S, S') \\
Pre{\leq_n}(S) \ \equiv\ \forall\, p \leq n.\ Pre_p(S) \quad \text{and} \quad Post{\leq_n}(S) \ \equiv\ \forall\, p \leq n.\ Post_p(S)
\end{array}
$$

[1] For concise notation, we typically write process identifiers of procedures/functions as indices rather than as parameters.

With these prerequisites, we can state the following RG decomposition rule for $\texttt{SPAWN}_n$.

Theorem 6.1 (RG Rule for n + 1 Interleaved Components)
The following RG decomposition rule for n + 1 interleaved components is correct.

$$\frac{\begin{array}{l}(1) \vdash G_p(S,S)\\ (2) \vdash R_p(S,S_0) \wedge R_p(S_0,S_1) \rightarrow R_p(S,S_1)\\ (3) \vdash G_p(S,S_0) \rightarrow R_q(S,S_0)\\ (4) \vdash Pre_p(S) \wedge R_p(S,S_0) \rightarrow Pre_p(S_0)\\ (5) \vdash Post_p(S) \wedge R_p(S,S_0) \rightarrow Post_p(S_0)\\ (6)\ Inv(S), Pre_p(S) \vdash [\langle R_p, G_p, Inv(S), \texttt{PROC}_p(S)\rangle]\ Post_p(S)\\ (7) \vdash (\exists\ S.\ Init(S)) \wedge (Init(S) \rightarrow Inv(S) \wedge Pre{\leq_n}(S))\end{array}}{Init(S) \vdash [\langle R{\leq_n}, G{\leq_n}, Inv(S), \texttt{SPAWN}_n(S)\rangle]\ Post{\leq_n}(S)}$$

Proof The proof is by structural induction over n (which is a special case of rule (2.3)). The case for $n = 0$ follows with premise (6) for $p = 0$. In the induction step, we use premise (6) as a contract for process $n+1$ and the induction hypothesis as a contract for the remaining interleaved components with identifiers at most n, respectively. Then the proof (essentially) applies rule 5.2 where the first component is now process $n + 1$ (with RG conditions Pre_{n+1}, R_{n+1}, ...) and the second component is the rest of the system with processes $p \leq n$ (and RG conditions $Pre{\leq_{n+1}}$, $R{\leq_{n+1}}$, ...). This is possible due to our higher-order logic setting where theorems are parametrized with higher-order-logic (RG) predicate functions. □

Rule 6.1 is sound for both partial and total correctness assertions under weak-fair/unfair interleaving. It is not difficult to extend it with a predicate $Runs_p$: *state* → *bool*, as explained in Section 5.3, to encompass the compositional verification of absence of deadlock.

6.1.1. Compositional Verification of a Parallel Sieve Algorithm

Now we illustrate the application of rule 6.1 to verify the total correctness of a standard example: It is a parallel algorithm SIEVE [54] that calculates all prime numbers up to a given bound based on the well-known Sieve of Eratosthenes.

The Sieve Algorithm in RGITL. Figure 6.1 shows the algorithm in the abstract programming language of RGITL. The algorithm $\texttt{SIEVE}_{\lceil\sqrt{(\mathsf{MAX})}\rceil-2}$ computes all prime numbers from 2 to constant MAX: *nat*. This is achieved by removing multiples of the numbers from 2 to $\lceil\sqrt{(\mathsf{MAX})}\rceil$ using parallel operations $\texttt{REM}_p(Nf)$. Natural numbers are represented by a boolean function Nf: *nat* → *bool* such that a number n is part of Nf if and only if $Nf(n)$ is *true*. Hence, an operation $\texttt{REM}_p(Nf)$ of process p removes the multiples $2 * (p + 2)$, $3 * (p + 2)$, ... from Nf by setting the corresponding Nf-slot to *false* where Nf is initially $\lambda\ n.\ true$.[2]

[2] We use an offset of 2 in the specifications, since process identifiers are 0-based in the parallel decomposition rule.

```
SIEVE_n(Nf) {                          REM_p(Nf) {
  if* n = 0 then                         let K = 2 in {
    REM_0(Nf)                              while K ≤ (MAX/(p + 2)) do {
  else                                       Nf(K ∗ (p + 2)) := false;
    REM_n(Nf) ||_nf SIEVE_{n−1}(Nf)          K := K + 1
}                                          } } }
```

Figure 6.1.: Specification of SIEVE in RGITL.

The Total Correctness of SIEVE. For concise specification of the total correctness theorem for SIEVE, we first define the following relevant sets of natural numbers, similar to [54]. The set $mults(n)$ of a given number n is the set of n-multiples k that are not bigger than MAX:

$$k \in mults(n) \leftrightarrow k \leq \mathsf{MAX} \wedge \exists\ m.\ m \geq 2 \wedge k = m * n$$

Similarly, the set $nats(nf)$ is the set of natural numbers k that are greater or equal 2 and not bigger than MAX for a given representation function Nf.[3]

$$k \in nats(Nf) \leftrightarrow (2 \leq k \leq \mathsf{MAX}) \wedge Nf(k)$$

Finally, the function $\lceil\sqrt{(n)}\rceil$ is specified with two axioms:

$$\lceil\sqrt{(n)}\rceil * \lceil\sqrt{(n)}\rceil \geq n$$
$$\lceil\sqrt{(n)}\rceil \neq 0 \rightarrow (\lceil\sqrt{(n)}\rceil - 1) * (\lceil\sqrt{(n)}\rceil - 1) < n$$

With these prerequisites, we can state the central total correctness theorem for SIEVE:

$$\begin{aligned} & Nf = \lambda\ n.\ true \\ \vdash\ & \langle Nf' = Nf'', true, true, \mathtt{SIEVE}_{\lceil\sqrt{(\mathsf{MAX})}\rceil - 2}(Nf)\rangle \\ & \quad \forall\ k.\ 2 \leq k \leq \mathsf{MAX} \rightarrow (prime(k) \leftrightarrow k \in nats(Nf)) \end{aligned} \tag{6.1}$$

where predicate $prime(k)$ defines k to be a prime number in the standard way.

The RG Instances for SIEVE. The proof of (6.1) instantiates rule 6.1. The system state S is simply Nf, the parameter procedure $\mathtt{PROC}_p$ is $\mathtt{REM}_p$ and the predicates of the

[3] The definitions of *mults* and *nats* are non-recursive, but it is not difficult to give equivalent recursive versions.

rule are instantiated as follows:

$$
\begin{aligned}
&\mathit{Init}(\mathit{Nf}) \equiv \mathit{Nf} = \lambda\ n.\ \mathit{true}\\
&\mathit{Pre}_p(\mathit{Nf}) \equiv \mathit{true}\\
&R_p(\mathit{Nf}', \mathit{Nf}'') \equiv \mathit{nats}(\mathit{Nf}'') \subseteq \mathit{nats}(\mathit{Nf}')\\
&G_p(\mathit{Nf}, \mathit{Nf}') \equiv \mathit{nats}(\mathit{Nf}') \subseteq \mathit{nats}(\mathit{Nf}) \wedge (\mathit{nats}(\mathit{Nf}) \setminus \mathit{nats}(\mathit{Nf}')) \subseteq \mathit{mults}(p+2)\\
&\mathit{Inv}(\mathit{Nf}) \equiv \forall\ n.\ n \leq \mathsf{MAX} \wedge \mathit{prime}(n) \rightarrow n \in \mathit{nats}(\mathit{Nf})\\
&\mathit{Post}_p(\mathit{Nf}) \equiv\ \ \mathit{nats}(\mathit{Nf}) \cap \mathit{mults}(p+2) = \emptyset
\end{aligned}
$$

Proving the RG Assertion for $\texttt{REM}_p$. With these instances, the predicate logic premises of rule 6.1 are straight-forward. The instance of the central premise (6) for total correctness of $\texttt{REM}_p$ is:

$$\mathit{Pre}_p(\mathit{Nf}), \mathit{Inv}(\mathit{Nf}) \vdash \langle R_p, G_p, \mathit{Inv}(\mathit{Nf}), \texttt{REM}_p(\mathit{Nf})\rangle\ \mathit{Post}_p(\mathit{Nf})$$

The proof of this assertion is by symbolic execution of sequential programs according to Section 4.3 and induction over the variant $\mathsf{MAX} - K$ according to rule (2.3). We start by unfolding the call and executing the **let** . Then we generalize the local variable $K = 2$ to the following loop invariant

$$2 \leq K \wedge \forall\ k.\ 2 \leq k < K \rightarrow k * (p+2) \notin \mathit{nats}(\mathit{Nf})$$

which states that k-multiples of $(p+2)$ where k is less than the current value of K have been already removed by $\texttt{REM}_p(\mathit{Nf})$.

Symbolically executing the while loop gives two cases: For the first case the loop exits immediately and the post-condition holds by the definition of the rely and post-condition predicates. The second case steps through the loop. The preservation of the guarantee and invariant by the two assignments is not difficult to prove by just unfolding the predicate logic definitions. Finally, the case where the loop is reiterated is closed with the induction hypothesis.

$\texttt{MUTEX}(\mathit{Labelf}, N)$ {
  $\texttt{MUTEX}_0(\mathit{Labelf}, N) \parallel_{\mathsf{nf}} \ldots \parallel_{\mathsf{nf}} \texttt{MUTEX}_n(\mathit{Labelf}, N)$
}

$\texttt{MUTEX}_p(\mathit{Labelf}, N)$ {
  while *true* **do** {
    await $N > 0$;
    $N := 0, \mathit{Labelf}_p := \mathit{true}$;
    skip;
    $N := 1, \mathit{Labelf}_p := \mathit{false}$;
  } }

Figure 6.2.: General Mutex for n Processes.

6.2. State-Local RG Rule

In this section, we derive a *state-local* rule from rule 6.1. It simplifies the overall system state S: *state* with $n + 1$ local states (and the shared state) to a state with just 2 designated local states (plus the shared state). Using such state-local rules is beneficial when verifying symmetric concurrent systems where the constituent components exhibit similar behaviors. (We show a challenging application in the third part of this work where we also combine this rule with our proof methods for linearizability and lock-freedom.) We have mechanized the derivation in KIV.

6.2.1. Motivation

To better understand why state-local rules can be useful, consider the general version of the simple mutex algorithm as shown in Figure 6.2. It considers an arbitrary finite number of parallel processes that repeatedly try to enter a critical section: Synchronization is again via a shared counter N that is initially 1 and set to 0 by the process that enters the critical section. Instead of the two explicit local boolean labels L_1 and L_2, the generic algorithm uses a higher-order function Labelf: $\mathit{nat} \to \mathit{bool}$ that models an arbitrary number of process-local labels. Initially, the function is $\lambda\ p.\ \mathit{false}$. Each process $p \leq n$ works on its label Labelf_p exclusively.

We could use rule 6.1 to prove the mutual exclusion property for this algorithm. The rule's state variable S would be instantiated with Labelf, N. Then lifting the central mutual exclusion invariant for 2 processes, i.e., $\neg\ L_1 \vee \neg\ L_2$, to an arbitrary number of processes gives:

$$\forall\ p.\ \mathit{Labelf}(p) \to \forall\ q \neq p.\ \neg\ \mathit{Labelf}(q)$$

This simple example already shows that with rule 6.1, specifications are typically based on process functions and quantification over process identifiers. Consequently, they are less concise and the verification requires more involved reasoning that is harder to automate. These complications can often be avoided: For the mutex algorithm, it is indeed sufficient to consider a pair of two representative processes and to prove correctness for this pair only. This is due to the inherent symmetry of the locking

mechanism for each process. Our state-local rule below makes this simplification for (symmetric) concurrent systems with an arbitrary finite number of processes possible.

The use of state-local RG rules is not always necessary. For instance, in Section 6.1, we have shown the correctness of the parallel `SIEVE` algorithm for an arbitrary number of processes using rule 6.1, but without referring to any local state information in the specifications/proofs. These only consider the shared state Nf. In that case, avoiding to talk about the local counters K in the RG specifications relies on the ability to establish local invariants during symbolic execution only, instead of encoding them in the invariant predicate. Further proof techniques such as ownership annotations, see Section 12.1, can also help to avoid explicit reasoning about local states (basically by making relevant local information globally visible). However, while the simplicity of just referring to the shared state in RG specifications is of course desirable, it is hard to achieve (or even impossible) in some cases.

For the general mutex algorithm, we can not express the mutual exclusion property without referring to local labels of different processes in the specifications. A more complex example is property *ishazard* in Section 12.2. In such cases, the state-local RG rule below makes specification and verification of a system with an arbitrary finite number of processes just as simple as it would be for two processes.

6.2.2. The State-Local RG Rule

First of all, we split the overall system state S of rule 6.1 into its local parts and the shared part, respectively. Hence, the state variable S of the global rule now becomes a tuple $LSf, \mathcal{S}$ where the higher-order function $LSf\colon nat \to lstate$ models all local states and $\mathcal{S}\colon sstate$ is the shared state. For our designated two local states we will use variables $LSp\colon lstate$ and $LSq\colon lstate$ in the following, where each local state LS includes a process identifier with selector function $LS.id$.[4]

As an additional specification predicate, we introduce predicate D of type $lstate \times lstate$ which encodes disjointness properties between the two local states. With these prerequisites, we first introduce the central proof obligation of the state-local rule, then we give the rule and prove its correctness. The central state-local proof obligation is:

$$\begin{aligned} & Pre(LSp, \mathcal{S}), Inv(LSp, \mathcal{S}), Inv(LSq, \mathcal{S}), D(LSp, LSq) \\ \vdash\ & [\langle R(LSp', \mathcal{S}', \mathcal{S}'') \wedge LSp' = LSp'', \\ & \quad G(LSp, LSq, \mathcal{S}, LSp', \mathcal{S}'), \\ & \quad Inv(LSp, \mathcal{S}) \wedge Inv(LSq, \mathcal{S}) \wedge D(LSp, LSq), \\ & \quad \texttt{PROC}(LSp, \mathcal{S})\rangle]\ Post(LSp, \mathcal{S}) \end{aligned} \tag{6.2}$$

Compared with premise (6) of rule 6.1, the program and environment transitions now maintain an invariant $Inv(LSp, \mathcal{S})$ for the current process p, a further invariant $Inv(LSq, \mathcal{S})$ for the other process q and the disjointness predicate D between the

[4] Tacitly, we assume that $LSp.id \neq LSq.id$ for two local states without mentioning this explicitly in the specifications for brevity.

local states of these two representative processes. The rely additionally assumes that the local state LSp of the current process is not concurrently changed.

Theorem 6.2 (The State-Local RG Rule)
The following state-local RG rule is correct.

$$\frac{\begin{array}{ll}
(1) & \vdash G(LSp, LSq, \mathcal{S}, LSp, \mathcal{S}) \\
(2) & \vdash R(LSp, \mathcal{S}, \mathcal{S}_0) \wedge R(LSp, \mathcal{S}_0, \mathcal{S}_1) \rightarrow R(LSp, \mathcal{S}, \mathcal{S}_1) \\
(3) & \vdash G(LSp, LSq, \mathcal{S}, LSp_0, \mathcal{S}_0) \rightarrow R(LSq, \mathcal{S}, \mathcal{S}_0) \\
(4) & \vdash Pre(LSp, \mathcal{S}) \wedge R(LSp, \mathcal{S}, \mathcal{S}_0) \rightarrow Pre(LSp, \mathcal{S}_0) \\
(5) & \vdash Post(LSp, \mathcal{S}) \wedge R(LSp, \mathcal{S}, \mathcal{S}_0) \rightarrow Post(LSp, \mathcal{S}_0) \\
(6) & \vdash D(LSp, LSq) \rightarrow D(LSq, LSp) \\
(7) & (6.2) \\
(8) & \vdash \quad (\exists\ LSf, \mathcal{S}.\ Init(LSf, \mathcal{S})) \\
 & \quad \wedge\ (Init(LSf, \mathcal{S}) \rightarrow \\
 & \qquad (\forall\ p.\ Init(LSf_p, \mathcal{S}) \wedge Pre(LSf_p, \mathcal{S}) \wedge Inv(LSf_p, \mathcal{S})) \\
 & \qquad \wedge\ (\forall\ p \neq q.\ Init(LSf_p, \mathcal{S}) \wedge Init(LSf_q, \mathcal{S}) \rightarrow D(LSf_p, LSf_q)))
\end{array}}{Init(LSf, \mathcal{S}) \vdash [\langle R{\leq_n}, G{\leq_n}, Inv(LSf, \mathcal{S}), \texttt{SPAWN}_n(LSf, \mathcal{S})\rangle]\ Post{\leq_n}}$$

Premises. Premises (1) - (6) are self-explanatory and premise (7) has already been described above. The initialization condition (8) does not use local state only but talks about all local states LSf. However, this condition is typically trivial to specify/verify in applications. Moreover, we have overloaded predicate names to avoid extra syntax, e.g., the global predicate $Init(LSf, \mathcal{S})$ is overloaded with its state-local version $Init(LS, \mathcal{S})$. Further overloadings are used in the conclusion of the rule:

Conclusion. The global invariant $Inv(LSf, \mathcal{S})$ states that the state-local invariants and disjointness properties hold for all local states.

$$Inv(LSf, \mathcal{S}) \equiv \forall\ p \neq q.\ Inv(LSf_p, \mathcal{S}) \wedge D(LSf_p, LSf_q)$$

The guarantee $G{\leq_n}$ ensures that each system transition preserves some state-local guarantee $G(LSf_p, \dots)$ and that it does not modify the local state of other processes q.

$$G{\leq_n} \equiv \exists\ p \leq n.\ \forall\ q \neq p.\ G(LSf_p, LSf_q, \mathcal{S}, LSf'_p, \mathcal{S}') \wedge LSf_q = LSf'_q$$

Similarly, the global rely $R{\leq_n}$ states that the system's environment preserves all state-local rely conditions and that it does not modify local states.

$$R{\leq_n} \equiv \forall\ p \leq n.\ R(LSf'_p, \mathcal{S}', \mathcal{S}'') \wedge LSf'_p = LSf''_p$$

In the remainder of this section, we sketch the soundness proof of rule 6.2. It applies rule 6.1 for specific instances of its parameters. These are explained in the following:

The global procedure $\texttt{PROC}_p(S)$ is instantiated with the state-local procedure $\texttt{PROC}(LSf_p, \mathcal{S})$. A suitable instance for predicate G_p of rule 6.1 is

$$\forall\ q \neq p.\ G(LSf_p, LSf_q, \mathcal{S}, LSf'_p, \mathcal{S}') \wedge LSf_q = LSf'_q$$

and similarly, we instantiate R_p with

$$R(LSf'_p, \mathcal{S}', \mathcal{S}'') \wedge LSf'_p = LSf''_p$$

With these instances, premises (1) - (3) of rule 6.1 hold. The pre-condition and post-conditions from the global rule simply correspond to $Pre(LSf_p, \mathcal{S})$ and $Post(LSf_p, \mathcal{S})$, respectively. Hence, premises (4) and (5) of the global rule also hold. The initialization predicate is simply $Init(LSf, \mathcal{S})$ and thus the global premise (7) holds as well.

It remains to prove the central premise (6) of the global rule with the given instances, which reduces to a proof of the following sequent:

$$\forall\ q \neq p.\ RG(7)^{LSf_p, LSf_q}_{LSp, LSq}, \Box\ (\forall\ q \neq p.\ LSf_q = LSf'_q), Inv(LSf, \mathcal{S}) \vdash RG(6)$$

The first formula in the antecedent is the RG assertion from (6.2) where LSp and LSq are substituted with LSf_p and LSf_q, respectively, and the resulting formula is strengthened to hold for an arbitrary other process $q \neq p$. The second formula can be syntactically derived, since the state-local procedure only works on LSf_p. The state-local invariants and the pre-condition are given initially by the pre-conditions of the global rule. To enable an inductive proof, however, we have dropped the global pre-condition already from the antecedent, such that only the global invariant remains. The formula in the succedent is the RG assertion of premise (6) of rule 6.1 with the instances above for its predicates.

Then the proof first unfolds the definition of both RG assertions and weakens both undeclared procedures from the antecedent, since these can not be symbolically executed. In case of a total correctness assertion, an extra case results for the liveness part which can be easily shown. (Note that the global rely conditions and invariants imply the state-local ones for process p.) It remains to prove the safety part:

First we apply induction rules (4.3) and then (2.4) on the resulting $\xrightarrow{+}$ formula in the succedent to get a counter N for induction (2.3). Then the proof applies symbolic execution according to Theorem 2.2 and the unwinding rule 4.2 for $\xrightarrow{+}$. A symbolic execution step thus yields three cases:

If the interval ends in the current state, then the initial goal simplifies to showing that the post-condition of process p holds, which follows immediately from the state-local post-condition for p. Otherwise, if the considered arbitrary interval prefix ends now, it remains to show that the state-local guarantee conditions (plus invariants) sustain the global ones. This is not very difficult as the state-local guarantee conditions are ensured for an arbitrary other process q. In the final third case, where the arbitrary prefix does not end yet, one must prove that the initial goal before the symbolic execution step is repeated in the current state. This is the case indeed, since the global rely conditions ensure the state-local ones and invariants are propagated by

both program and environment transitions. The proof thus concludes by applying the induction hypothesis.

The application of the state-local rule to prove the generic mutex algorithm correct is quasi equal to the proof of the simple mutex with just 2 processes (see Section 5.3): The local state LSp is instantiated with L_1 and LSq with L_2. The proof of premise (7) of the state-local rule corresponds to the proof of premises (6)/(12) of rule 5.2 for processes 1/2. Thus the state-local rule even better exploits the underlying symmetry than rule 5.2.

We have derived several versions of this state-local rule where, e.g., the global state is included in the disjointness predicate, or the local state of the other process is a parameter of the rely predicate, etc. These variants are not detailed here as they only slightly differ from the given rule. Instead, they are given online [59].

7. Related Work and Conclusion

Comparison with Own Previous Work

The Ph.D. thesis [6] studies a basic embedding of RG reasoning in RGITL. There, only RG assertions for partial correctness are considered which are directly expressed using the **unless** operator. [6] derives one basic decomposition rule (without pre- and post-conditions) for weak-fair n-interleaving. This work improves on this basic approach as follows:

- As a minor improvement, we now define the semantics of $\xrightarrow{+}$ directly as an **until** formula. More importantly, we introduce and implement new RG operators $[\ .\ ]$ and $\langle\ .\ \rangle$ for partial/total correctness which leads to more concise specifications (see Section 4.2).
- [6] verifies $\xrightarrow{+}$ formulas by directly applying Theorem 2.2. In contrast, we now use a Hoare-style calculus for the verification of RG assertions for sequential programs (Section 4.3) and support their verification with several heuristics for automation. This leads to more intuitive proofs that resemble proofs for sequential programs in dynamic logic which KIV supports too.
- Based on our new RG operators, we derive various rules for the verification of partial/total correctness and absence of deadlock of weakly-fair/unfair interleaved programs (Sections 5.1, 5.3 and 6.1).
- We derive state-local rules (Section 6.2) which lead to simpler specifications/proofs for symmetric concurrent algorithms.
- We modularize our proofs for n-interleaving by parameterizing RG rules for two-interleaving with higher-order predicate functions (see proof of rule 6.1).
- We evaluate our RG rules using standard examples that have not been considered in [6], by verifying total correctness of FIND, `SIEVE`, etc. (Various other new case studies are considered in Part III.)

Comparison with Other Approaches

Many different versions of RG rules for shared variable programs exist: RG reasoning for partial correctness of these programs goes back to [53, 63, 92, 109]. For programs with send/receive primitives, similar ideas can be already found in [71]. A systematic

overview and presentation of RG rules for shared variable programs is given in [109]. They provide sound and complete calculi for partial/total correctness and absence of deadlock. Thus their approach is closely related to ours and we focus on their work in the following:

Based on an Aczel-trace semantics, [109] define RG assertions for partial correctness of a program P as

$$P \ \underline{sat} \ (pre, rely, guar, post)$$

which (given *pre* for the initial state) roughly requires $\Box\ rely \rightarrow \Box\ guar$ to hold for *all prefixes* of an Aczel-trace and *post* upon termination. This translates to our setting as

$$pre \vdash [rely^*, guar, true, P]\ post$$

with a trivial invariant. The transitive closure * for rely steps is necessary here, since different from our approach, relies are not necessarily transitive in their setting. Based on this translation, we have derived the rules of their complete RG calculus in RGITL, except for their rule for auxiliary variables. Our derivation specifies a program P in their rules as an undeclared procedure `PROC` and premises for P are given as axioms $\texttt{PROC} \vdash \varphi$, respectively. Then deriving a rule of their calculus in RGITL, typically rewrites `PROC` with φ and then uses symbolic execution and induction (see Chapter 2) to derive the rule's conclusion.

A major difference to our approach is that [109] and others [53, 86, 92, 109] give semantic proofs for their RG rules on paper while we *mechanically derive* the soundness of our rules. Our soundness proofs for parallel decomposition rules directly reflect the possible executions that evolve from interleaving two RG assertions. [109] focuses on theoretical aspects of their calculi such as completeness whereas our main concern here is the derivation of compositional proof methods for linearizability/lock-freedom based on RG reasoning and the application of these methods to verify intricate algorithms correct (see Part III). The calculus for total correctness in [109] introduces an additional iteration rule to prove termination for loops. This rule can be derived in RGITL using induction rule (2.3) and symbolic execution. Finally, we have directly taken over their method for proving absence of deadlock that goes back to [92].

Mechanized soundness proofs for RG rules have been studied for the first time in [83] based on an encoding of the Aczel-trace semantics [109] in Isabelle/HOL. Their work considers partial correctness only. The derivation of RG rules in RGITL is simpler than in [83], since the proofs are not based on a semantic encoding of traces and programs in higher-order logic, but rather on a native implementation. Thus it only takes a moderate effort to derive different variants of RG rules in our setting. [83] considers only standard applications of their RG calculus.

Conclusion

In the second part of this work, we have introduced RG reasoning in RGITL. We have mechanically specified and verified various RG rules for both sequential and interleaved

programs and illustrated their application using standard examples. In particular, we have introduced two novel RG operators for partial/total correctness which we support with specific (Hoare-style) rules for symbolic execution. This leads to intuitive proofs for sequential programs, similar to dynamic logic proofs for sequential programs. Based on these new operators, we have derived various decomposition rules for weak-fair/unfair interleaved systems, including rules to prove the absence of deadlock and state-local rules for the important class of symmetric concurrent programs. This work underlies our proof methods for highly concurrent data structures that we introduce in the next part.

The main effort when specifying such rules is to find the right level of expressiveness. Typically, only when applying these rules to verify a concrete property one notices that, e.g., some parameter of a RG condition is missing. For instance, only when verifying the FIND algorithm we noticed that it can be beneficial to have two-state post-conditions instead of a mere single-state predicate. Similarly, only when verifying the hazard pointers case study (see Section 12.2) we noticed that it is beneficial to have a process identifier as part of the local state. Consequently, one has to define and verify new versions of a decomposition rule and to adapt the proofs for the case study. Such iterative steps constitute the main effort to improve the embedding of RG reasoning.

In future work, we want to introduce a RG rule for auxiliary variables to make our RG calculus for partial correctness complete w.r.t. the calculus in [109]. Ideally, we would prefer to derive the rule from a more general rule that works for a wider class of temporal formulas. Furthermore, our current rule from Section 5.3 implies that an interleaved system never runs into a deadlock which is a safety property. Finding a compositional proof method for the liveness property of deadlock freedom [49] that ensures global progress for lock-based systems under a sufficiently fair scheduler would be another option for future work.[1] Another possible direction is to automate the calculation of the stable part of a state formula over a rely condition, similar to [105]. Currently, this is a specific interactive step throughout the symbolic execution of a given program, as explained in Section 4.3. An automated generation could reduce the verification effort. Finally, having a convenient way to directly apply RG rules for parallel programs in proofs without instantiating a RG theory, is another option for future work.

[1] The notion of "deadlock freedom" in [109] corresponds to our global safety property "absence of deadlock". However, the term "deadlock freedom" in [49] denotes a global liveness property where some method eventually succeeds to acquire a lock under specific scheduling requirements.

Part III.

Verifying Highly Concurrent Data Structures in RGITL

In the second part of this work we have introduced RG reasoning in RGITL. The third part of this work now applies this theory in the domain of highly concurrent data structures. We derive novel RG-based proof methods for the compositional verification of the central global safety/liveness properties of linearizability and lock-freedom. Moreover, we illustrate the application of our methods to verify several challenging concrete implementations correct some of which had no (mechanized) verification before. The soundness proofs of our rules (theorems) and their application to verify concrete case studies correct, are mechanized in KIV; we refer the reader to respective online descriptions throughout the text.

In particular, we focus on the verification of highly concurrent implementations that use fine-grained locking or avoid the use of locks and apply mutual helping schemes to maximize the throughput of concurrent accesses on the data structure. Variants of some of the algorithms that we consider here [69, 70] are, e.g., part of Java's concurrency package [49]. We also consider specific memory management techniques that are applicable to various implementations in environments without garbage collection and explicitly deal with fundamental problems that arise in such a setting.

The structure of part III is as follows: In Chapter 8, we motivate concurrent data structures for multi-core computers and informally introduce their central notion of correctness, namely linearizability. Then we give two proof methods for linearizability in Chapters 9 and 10, respectively, and illustrate their application to verify two non-trivial concurrent multiset implementations (a wait-free implementation and one that uses fine-grained locking). Next, Chapter 11 defines a proof method for the liveness property of lock-freedom and illustrates its application to verify liveness for a highly concurrent set data structure. Chapter 12 illustrates further applications of our proof methods by verifying further lock-free data structures with explicit concurrent garbage collection. Finally, we conclude in Chapter 13 with a discussion of related and possible future work.

8. Highly Concurrent Data Structures

This chapter first informally introduces the basic domain-specific concepts of highly concurrent data structures (Section 8.1) with three simple example implementations (Section 8.2). Then Section 8.3 intuitively describes their central correctness notion of linearizability along with its well-known proof method of linearization points. (These concepts are then formalized in the next chapter.)

8.1. Introduction

Multi-core architectures have become ubiquitous in our computers. Thus data structure implementations that offer fast concurrent access on these machines are of particular importance. Over the last decades, a variety of efficient implementations of standard data structures have been developed for these architectures [14, 45, 69, 72, 102]. Instead of locking entire data structure operations (coarse-grained locking), these implementations typically either use fine-grained locking schemes that only lock small parts (e.g., individual memory locations), or non-blocking techniques that avoid the use of locks and use hardware instructions such as CAS (compare-and-swap) or LL/SC (load linked/store conditional) instead. Thus a higher degree of parallelism can be achieved. Such data structures are used for instance in operating system kernels or in libraries of common programming languages, e.g., java.util.concurrent for Java, etc.

Highly concurrent data structures are typically developed to meet certain correctness (safety or liveness) conditions. Several such properties have been proposed [49], but two have gained particular attention over the last years:

- Linearizability [48] is a central safety property: Roughly speaking, a data structure is linearizable if each of its concurrent behaviors corresponds to some sequential behavior of an abstract data type with atomic operations. Furthermore, linearizability imposes the following intuitive constraint on the order of corresponding abstract behaviors: It must preserve the execution order of concrete operations that do not overlap in time.

- Lock-freedom [67] is an essential liveness property of non-blocking concurrent data structures. It excludes both system-wide livelocks and deadlocks of a data structure even in the presence of indefinite delays of individual processes that access it.

The subtlety of these algorithms that results from the multitude of possible interleaved executions, leads us to formal specification/verification to gain a more precise

```
INC(N): nat
  var m, n: nat;
  n := N;
  m := n + 1;
  N := m;
  return n
```

Figure 8.1.: Incorrect Implementation of a Shared Counter.

understanding and more confidence in their correctness [18, 25]. However, the formal specification and verification of these algorithms is difficult. A number of approaches have been proposed for proving linearizability and several implementations have been verified linearizable, e.g., [3, 13, 26, 48, 103]. Similarly, several approaches for proving lock-freedom have been introduced and applied, e.g., [17, 38, 89].

The formal methods proposed in the literature for verifying highly concurrent data structures use a variety of techniques ranging from separation logic [85], temporal logic [82], rely-guarantee reasoning [53], shape analysis [87] or model checking [10]. Our proof methods for linearizability and lock-freedom that we have derived in RGITL are essentially based on temporal logic (as defined in Part I) and RG reasoning (as introduced in Part II). Occasionally, we also apply additional techniques such as ownership annotations [15, 76] or separation logic to simplify specifications and proofs, briefly introducing these additional techniques where necessary.

8.2. Three Simple Data Structures

This section explains the basic concepts of highly concurrent data structure implementations using three simple examples: a counter, a stack [102] and a queue [20]. In the basic setting, an arbitrary finite number of processes can invoke the data structure's access methods concurrently.

8.2.1. A CAS-Based Counter

First we consider a simple counter data structure $N\colon nat$ which offers an increment method $\texttt{INC}(N)\colon nat$ that increments N by 1 and returns the value of N before the increment.

Figure 8.1 shows a possible implementation of method $\texttt{INC}(N)$. Assuming that the shared counter cannot be incremented atomically (i.e., such that no other process can observe a partial update), the algorithm first (atomically) reads N into a local variable n, then it increments n locally and (atomically) sets N to the new value m afterwards. Unfortunately, this implementation exhibits the following undesired behavior: When two concurrent increments both read the same counter value, variable N is incremented just once instead of twice as one would expect when two increment operations are executed.

```
INC(N) : nat
  var m, n: nat;
  lock(N);
    n := N;
    m := n + 1;
    N := m;
  unlock(N);
  return n
```

Figure 8.2.: Shared Counter Implementation with Coarse-Grained Locking.

To exclude this undesired behavior, both the reading and the update of N can be protected by a lock as Figure 8.2 shows. Sadly, such coarse-grained locking has two disadvantages:

1. It downgrades parallelism by enforcing a coarse-grained sequential order on concurrent data structure accesses.

2. The crash of a process that currently holds a lock can cause a deadlock and similarly, the preemption of such a process can lead to unpredictable delays of other processes.

Highly concurrent algorithms try to only lock small parts (which improves on the first problem above), or even not to use locks at all (which solves the second problem above). In the latter case, they apply atomic synchronization instructions such as CAS, typically in combination with the following optimistic try and retry programming scheme:

1. Store the relevant part of the shared data structure in a local variable (called a "snapshot").

2. Locally prepare a new version of the shared part to be updated.

3. Update the shared data structure atomically (using CAS) if no interference has occurred since the snapshot was taken. Otherwise, go to Step 1.

If another process changes the relevant shared part while the current process executes Step 2, it is forced to retry its update (in Step 3) by starting with Step 1 again. Consequently, algorithm designers try to keep Step 2 as small as possible to minimize the probability of having to repeat Steps 1 – 3 due to a deprecated snapshot.

The CAS instruction that Step 3 uses, compares whether a shared location *Now* still holds the previous snapshot value *old* (taken in Step 1). If the values are equal, then *Now* is updated to a new value *new* (computed in Step 2) and true is returned (i.e., Step 3 succeeds); otherwise false is returned (i.e., Step 3 fails and Step 1 must be executed again).

```
INC(N) : nat
  var m, n: nat;
  do
    n := N;
    m := n + 1;
  until CAS(N, m, n);
  return n
```

Figure 8.3.: A CAS-Based Shared Counter Implementation.

```
atomic CAS(Now, new, old): bool
    if (Now = old) {
      Now := new; return true
    } else return false
```

The execution of `CAS` is guaranteed to be atomic by the underlying machine, so it appears to execute in one indivisible step. CAS instructions are supported by many processors such as Intel-x86, ... and programming languages such as C/C++, ...

Figure 8.3 shows the application of the basic lock-free programming scheme above to implement a shared counter in a lock-free manner. Of course, this basic scheme can be extended in lots of different ways to further improve parallelism, e.g., by introducing mutual helping/elimination [69, 72].

8.2.2. A CAS-Based Stack

This section informally describes a second and slightly more complex example of a lock-free data structure, a stack with push and pop operations that can be invoked concurrently by a finite number of processes. The implementation is attributed to Treiber [102]; Figure 8.4 shows the pseudo code.[1]

The stack is represented in memory as a singly linked list of nodes (with selectors .el/.nxt for the element/next pointer) that is accessed via a shared top-of-stack pointer *Top*.

Whenever a process executes a push operation `PUSH`, it first allocates a new node and initializes it with the input element a. Then it repeatedly tries to CAS the top-of-stack pointer to this new node using the lock-free scheme above: This includes taking a local copy of the shared top pointer and setting the new node's next reference to this snapshot. The CAS instruction then atomically tests if the shared top pointer is still equal to the snapshot. If this is the case it succeeds by updating *Top* to the new node; otherwise it fails and the loop is reiterated.

[1] In pseudo code instructions we typically leave heap access implicit.

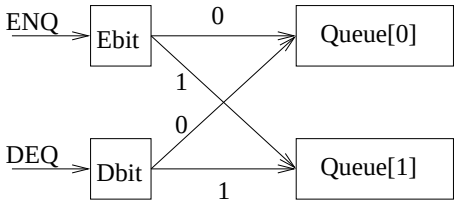
```
record node = {.el: elem, .nxt: ref};
var Top: ref(node);

PUSH(a: elem)                          POP() : empty | elem
  var top, node: ref;                    var top, nxt: ref;
  node := new node();                    do
  node.el := a;                            top := Top;
  do                                       if (top = null) {
    top := Top;                              return empty
    node.nxt := top;                       }
  until CAS(Top, node, top)                nxt := top.nxt;
                                         until CAS(Top, nxt, top);
                                         return top.el
```

Figure 8.4.: Treiber's Lock-Free Stack.

The pop operation `POP` proceeds similarly. It first reads the shared top: If the snapshot is null, the special value *empty* is returned to indicate an empty stack. Otherwise it locally stores the snapshot's next reference which becomes the target of the subsequent CAS instruction. If the CAS instruction succeeds, the top node is removed from the stack and its element is returned.

8.2.3. A Concurrent Queue Algorithm

Finally, we introduce a simple concurrent queue algorithm [20]. The queue implementation is based on two diffraction trees [2], one for the enqueue operations and the other one for the dequeue operations. The root of the enqueue/dequeue tree consists of a bit *Ebit*/*Dbit*, while each leaf points to a concurrent queue data structure *Queue*[0] and *Queue*[1]:



We assume that each of the two underlying queues meets the following two criteria: i) It ensures an atomic FIFO queue behavior and ii) if a dequeue operation encounters an empty queue, then it waits until it finds an element to dequeue instead of returning *empty*.

The overall queue data structure has two operations `ENQ` and `DEQ`. Figure 8.5 shows their implementation: `ENQ` first reads and tries to flip *Ebit* in a CAS-loop. Then it follows the upper/lower branch of the enqueue tree according to its snapshot *lbit* and

```
ENQ(a: elem)
  var lbit: bool;
  do
    lbit := Ebit;
  until CAS(Ebit, 1-lbit, lbit);
  Enq(Queue[lbit], a)
```

```
DEQ(): elem
  var lbit: bool;
  do
    lbit := Dbit;
  until CAS(DBit, 1-lbit, lbit);
  return Deq(Queue[lbit])
```

Figure 8.5.: A Concurrent Queue with 2 Diffraction Trees.

enqueues the input element a to $Queue[lbit]$ using the Enq method. The dequeue operation works similarly based on the Deq method.

The flip of $Ebit/Dbit$ ensures that the next enqueue/dequeue operation works on the other queue. This can reduce contention on the underlying queues and improve the performance of the overall data structure [2]. This algorithm significantly differs from the previous counter and stack data structures in terms of correctness, as we explain in the next section.

8.3. The Informal Notion of Linearizability

In this section we informally introduce linearizability [48], the quasi standard correctness condition for highly concurrent data structures. Basically, linearizability requires that each concrete behavior of a data structure implementation corresponds to some behavior of an abstract version of the data structure. This abstract version defines the semantics of the implementation in terms of an abstract state and atomic operations.

For instance, the abstract specification of the shared counter is simply a natural number N with an algebraic read and increment relation

$$AInc(N, N', Out) \equiv N' = N + 1 \wedge Out = N$$

where the natural numbers N/N' denote the value of the abstract state before/after an atomic transition of $AInc$ that computes an output Out.

Linearizability defines the behaviors of concrete and abstract operations in terms of *execution histories* which are finite sequences of events. An event models either the invocation inv or the return ret of a particular operation.

$$event\colon\ inv(nat, index, input) \mid ret(nat, index, output)$$

Each operation has an index of type $index$ and is invoked by some process $p\colon nat$ with some input value of type $input$, possibly returning an output value of type $output$. Each concrete operation corresponds to one abstract operation. In the following, we write the process identifier of an event e as an index e_p instead of a parameter $e(p, \ldots)$ for brevity.

Abstract histories evolve from executing sequences of abstract operations. Notice that abstract histories are always *sequential*, i.e., an invoke event is immediately followed by a return event of the same process. This is because abstract operations are

just atomic relations on the abstract state. In contrast, an invoke event of process p in a concrete history is typically followed by events of other processes due to the possible preemption of p.

Example 8.1 (A Non-Linearizable History)
The following history H represents an execution of the erroneous counter implementation from Figure 8.1, where processes p and q concurrently try to increment the counter N (initially 0), but one increment gets lost (irrelevant inputs/outputs are denoted as _):

$$H \equiv inv_p(inc,\ \_),\ inv_q(inc,\ \_),\ ret_p(inc,\ 0),\ ret_q(inc,\ 0)$$

It is easy to see that H can not be mapped to any equivalent history of the abstract counter specification: Starting with $N = 0$ and reordering the events in H to get an abstract execution history gives either

$$inv_p(inc,\ \_),\ ret_p(inc,\ 0),\ inv_q(inc,\ \_),\ ret_q(inc,\ 0)$$

or

$$inv_q(inc,\ \_),\ ret_q(inc,\ 0),\ inv_p(inc,\ \_),\ ret_p(inc,\ 0)$$

but both histories are not admissible histories of the abstract counter since the last return event has output 0 while it must be 1 according to the abstract relation AInc. That is, history H has no corresponding abstract behavior and thus the erroneous counter is not linearizable w.r.t. its abstract semantics.

The next example shows a linearizable history of the CAS-based counter from Figure 8.3.

Example 8.2 (A Linearizable History)
The following history

$$H = inv_p(inc,\ \_),\ inv_q(inc,\ \_),\ ret_p(inc,\ 1),\ ret_q(inc,\ 0)$$

reflects a possible behavior of the CAS-based counter where process q invokes an increment operation after process p and overtakes p by executing its increment first, thus returning 0*. It is simple to see that H can be reordered to the following sequential history of the abstract counter*

$$inv_q(inc,\ \_),\ ret_q(inc,\ 0),\ inv_p(inc,\ \_),\ ret_p(inc,\ 1)$$

where first q's increment takes effect and then p's. Thus H is linearizable.

Since all concurrent histories of the CAS-based counter have a corresponding abstract history, the implementation is said to be linearizable. Next reconsider the queue algorithm from Figure 8.5.

Example 8.3 (A Non-Linearizable Queue)
The queue implementation from Figure 8.5 is not linearizable. Let

$AEnq(In, Queue, Queue') \equiv Queue' = Queue + In$
$ADeq(Queue, Queue', Out) \equiv Queue \neq [\,] \wedge Queue' = Queue.rest \wedge Out = Queue.first$

define the abstract semantics of the concrete queue operations `ENQ` *and* `DEQ`*, where Queue is an algebraic list of elements that defines the abstract state with atomic relations AEnq/ADeq. Then the following concurrent queue history is not linearizable:*

$inv_p(deq, \_), inv_q(enq, a), ret_q(enq, \_), inv_q(enq, b), ret_q(enq, \_),$
$inv_q(deq, \_), ret_q(deq, b), inv_q(deq, \_), ret_q(deq, a), inv_q(enq, c), ret_q(enq, \_),$
$ret_p(deq, c)$

In the execution history above, dequeue process p is preempted right after flipping the Dbit. Then some other process q sequentially executes two complete enqueue operations, two complete dequeue operations and another enqueue, respectively. Now remember that linearizability requires that the order of non-overlapping executions is preserved in a corresponding abstract execution history. Hence, the order of operations of process q must not be changed when we construct an abstract history. However, the first dequeue of process q then violates the FIFO semantics of the abstract queue as it returns b even though a is the first element of Queue. Thus the implementation is not linearizable.

Interestingly, our queue example satisfies the correctness condition of *quiescence consistency* [49] which is strictly weaker than linearizability and allows for more reorderings in the construction of an abstract history.[2] In [20], we have formalized the notion of quiescent consistency and also introduced a proof method based on coupled simulations, but this is not in the scope of this thesis. For an overview of other correctness conditions see [20, 49].

8.3.1. Linearization Points

It is cumbersome to reason about linearizability by searching for a corresponding abstract behavior for each possible concrete behavior. More intuitive proof methods are desirable. One important proof method is based on the observation that linearizable data structure operations appear to take effect instantaneously at some point between their invocation and return [48]. This point is called the *linearization point*. It is a unique point in time when an abstract operation must be executed in a corresponding abstract execution. We say an operation *linearizes* or *takes effect* when it passes its linearization point. Hence, the unique order of linearization points in a linearizable concurrent execution, precisely determines the order of the atomic operations in the corresponding abstract sequential execution.

[2] The stack implementation based on diffraction trees as discussed in [91], p. 82, is not even quiescent consistent.

Reasoning about linearizability in terms of linearization points of an implementation is much more convenient than in terms of all concurrent executions. For instance, the CAS-based increment operation is linearizable: Its linearization point is its successful CAS instruction.

Similarly, we can reason about the correctness of the CAS-based stack from Figure 12.2. Its abstract semantics is

$$\begin{aligned} &APush(In, Stack, Stack') \equiv Stack' = In + Stack \\ &APop(Stack, Stack', Out) \equiv \\ &\quad \textbf{if } Stack = [\,] \textbf{ then } empty \textbf{ else } Out = Stack.\text{first} \wedge Stack' = Stack.\text{rest} \end{aligned} \tag{8.1}$$

where the abstract state is an algebraic list of elements *Stack* and the special value *empty* denotes the empty stack. With this semantics, we can easily identify the linearization points of the CAS-based stack implementation: These are its successful CAS instructions and when a pop operation returns *empty* it linearizes when it takes a null snapshot.

In the simple examples above, linearization points are internal, i.e., they coincide with one specific instruction of the running process. Moreover, they are static as they are independent from the concurrent execution of other processes. In more complex examples, it can not be statically determined whether an instruction is a linearization point, since this depends on the concurrent behavior of other processes. We call such linearization points potential linearization points [24]. The linearization of one process can also happen with an instruction of *another* process, called an external linearization point. We will verify case studies in the following that cover all possible types of linearization points. In the next chapter, we formally define linearizability and introduce a generic proof method that combines RG reasoning with the intuition of identifying linearization points.

9. A Generic Proof Method for Linearizability

This chapter first formally defines linearizability in terms of temporal logic, Section 9.1. Then Section 9.2 defines its well-known associated proof method of possibilities [48] that is based on the intuition of identifying linearization points. Afterwards we introduce our generic proof method that combines the idea of possibilities with RG reasoning in Section 9.3. Finally, Section 9.4 illustrates our proof technique using a novel wait-free and linearizable multiset implementation with intricate linearization points. We have mechanized the proof that our verification technique ensures linearizability according to the original definition [48] and that our multiset implementation is an instance of it. The predicate logic parts of the formalization of linearizability have been described in [23]. The presentation corresponds to our article [98].

9.1. Definition of Linearizability

9.1.1. The Abstract Specification

Linearizability defines the correctness of a concurrent implementation in terms of an abstract sequential specification. In general, the abstract specification consists of an abstract state of type *astate* that is initialized according to an initialization predicate *AInit* and it contains atomic abstract operation relations

$$AOP(I)(In, AS, AS', Out)$$

where I is the operation index. Each operation relation $AOP(I)$ defines the possible *atomic* transitions of the abstract state from AS to AS' with input and output values $In\colon input$ and $Out\colon output$, respectively.

Since linearizability compares histories of concrete and abstract specifications, we extend the abstract specification with sequential histories H_s. In this extended setting, a process with identifier $p\colon nat$ executes the history enhanced abstract atomic operation

$$\begin{aligned} &AOP_{p,H_s}(I)(AS, H_s, AS', {H_s}') \equiv \\ &\quad \exists\, In, Out. \quad AOP(I)(In, AS, AS', Out) \\ &\qquad\qquad \wedge\ {H_s}' = H_s + inv_p(I, In) + ret_p(I, Out) \end{aligned}$$

which executes $AOP(I)$ and additionally adds a pair of an invoke and return event to H_s.

Sequences of operations AOP_{p,H_s} that are executed by some process p from a finite set of processes, generate the histories of the abstract specification. In the remainder of this subsection we first introduce some simple functions on events and histories, and then we formally define the sequential histories of the abstract data type.

Definition 9.1 (Functions on Events and Histories)
The following simple functions are typically used with events/histories:

- *$e.p/e.i$ selects the process identifier/the operation index from an event e.*
- *For an invoke/return event e the values $e.in/e.out$ are the associated inputs and outputs.*
- *Predicates $inv?(e)$ and $ret?(e)$ test whether e is an invoke/return event.*
- *$\#\ H$ denotes the (finite) length of a history H.*
- *$H(n)$ is the n-th event for $n < \#\ H$.*
- *$H{\downarrow_p}$ is the projection of H to events e with $e.p = p$ (also defined similarly for sets of events).*

The histories of the abstract data type start in an abstract initial state according to predicate $AInit_{H_s}$ that requires $AInit$ to hold and H_s to be empty. Since the abstract operations are atomic, the resulting histories H_s are *sequential* according to the following definition.

Definition 9.2 (Sequential Histories)
A history H_s is sequential if predicate $seq(H_s)$ holds where

$$\begin{aligned} &seq(H_s) \equiv \exists\ m.\ \#\ H_s = 2m \wedge \forall\ n < m.\ irp(2n, 2n+1, H_s) \\ &\text{where } irp(m,n,H) \ \equiv\ \ m < n < \#\ H \wedge inv?(H(m)) \wedge ret?(H(n)) \\ &\qquad\qquad\qquad\qquad \wedge\ H(m).p = H(n).p \wedge H(m).i = H(n).i \end{aligned}$$

That is, sequential histories have an even length and consist of pairs of invoke and return events *irp* only.

Thus we can characterize the histories of the abstract specification with a simple recursive predicate *ahist*:

Definition 9.3 (The Histories of the Abstract Specification)
A sequential history H_s is a history of the abstract specification if predicate $ahist(H_s)$ holds where

$$ahist(H_s) \equiv \exists\ AS.\ linval(H_s, AS)$$

and predicate linval is recursively defined as

$$\begin{aligned} &linval([\,], AS) \equiv AInit(AS) \\ &linval(H_s + inv_p(I, In) + ret_p(I, Out), AS') \equiv \\ &\qquad \exists\ AS.\ linval(H_s, AS) \wedge AOP(I)(In, AS, AS', Out) \end{aligned}$$

In other words, for a given sequential history H_s, predicate *linval* inductively defines the abstract states AS that are reachable from an initial state (where $AInit$ holds) by executing the abstract operations $AOP(I)$ according to the invoke/return pairs in H_s.

9.1.2. The Concrete Specification

The concrete specification for linearizability is based on a concurrent system model

$$
\begin{array}{l}
\texttt{SPAWN}_{n,H}(CS)\ \{ \\
\quad \textbf{if*}\ n = 0\ \textbf{then} \\
\quad\quad \texttt{CPROC}_{0,H}(CS) \\
\quad \textbf{else} \\
\quad\quad \texttt{CPROC}_{n,H}(CS)\ \|_{\mathsf{nf}}\ \texttt{SPAWN}_{n-1,H}(CS) \\
\}
\end{array}
$$

which corresponds to $\texttt{SPAWN}_n$ from Section 6.1 as follows: We rename PROC to CPROC and the state variable $S\colon state$ to $CS\colon cstate$ to better discern concrete and abstract specifications. Moreover, we extend the state with a shared history variable H and use the following implementation for procedure $\texttt{CPROC}_{p,H}$:

$$
\begin{array}{l}
\texttt{CPROC}_{p,H}(CS)\ \{ \\
\quad \{\textbf{let}\ I, In, Out\ \ \textbf{in}\ \ \texttt{COP}_{p,H}(I, In; CS, Out)\}^* \\
\}
\end{array}
$$

That is, each spawned process p now repeatedly (note the iteration operator) executes the history enhanced concrete procedure $\texttt{COP}_{p,H}$ with some operation index $I\colon index$, input $In\colon input$ and output $Out\colon output$.

To prove linearizability, we implement the procedure $\texttt{COP}_{p,H}$ as follows: Each process p adds an invoke event $inv_p(I, In)$ to the shared history variable H before it executes the internal steps according to an undeclared procedure $\texttt{COP}_p$.

$$
\begin{array}{l}
\texttt{COP}_{p,H}(I, In; CS, Out)\ \{ \\
\quad H := H + inv_p(I, In); \\
\quad \texttt{COP}_p(I, In; CS, Out); \\
\quad H := H + ret_p(I, Out) \\
\}
\end{array}
$$

Upon termination of these internal steps that leave H unchanged, p adds a return event $ret_p(I, Out)$ to H. The initialization predicate $Init_H$ now requires H to be empty and $Init(CS)$ to hold in initial overall system states.

The visible behaviors H of $\texttt{SPAWN}_{n,H}$ are typically not sequential. Therefore, we introduce the more general notion of *legal* histories.

Definition 9.4 (Legal Histories)
A history H is legal, i.e., predicate legal(H) holds, if it consists of either i) pairs of

matching invoke and return events mp that correspond to terminated operation executions or ii) pending invoke events pi that do not have a matching return, corresponding to currently running operations.

$$
\begin{aligned}
&\mathit{legal}(H) \equiv \\
&\quad \forall\, n < \#\ H.\ \textbf{if}\ \mathit{inv}?(H(n))\ \textbf{then}\ ((\exists\ m.\ \mathit{mp}(n,m,H) \vee \mathit{pi}(n,H)) \\
&\qquad\qquad \textbf{else}\ \exists\ m.\ \mathit{mp}(m,n,H) \\
&\mathit{where}\ \mathit{mp}(m,n,H)\ \equiv \mathit{irp}(m,n,H) \wedge \forall\ m_0.\ m < m_0 < n \rightarrow H(m_0).p \neq H(m).p \\
&\mathit{and}\ \mathit{pi}(n,H)\ \equiv n < \#\ H \wedge \mathit{inv}?(H(n)) \wedge \forall\ m.\ n < m < \#\ H \rightarrow H(m).p \neq H(n).p
\end{aligned}
$$

The second conjunct of predicate *mp* above ensures that only events of *other* processes are acceptable between an invoke/return pair *irp*.

Example 9.1 (Legal Histories)
The history

$$\mathit{inv}_p(\mathit{push},\ a),\ \mathit{inv}_p(\mathit{pop},\ \_)$$

is not legal, since process p invokes a pop before its running push returns. In contrast, the history

$$\mathit{inv}_p(\mathit{push},\ a),\ \mathit{inv}_q(\mathit{pop},\ \_),\ \mathit{inv}_r(\mathit{pop},\ \_),\ \mathit{ret}_q(\mathit{pop},\ \mathit{empty})$$

is legal. It consists of two pending invokes of processes p and r and a matching pair of invoke and return events of process q.

The sequential histories H_s of the abstract specification and the histories H of $\texttt{SPAWN}_{n,H}$ are legal.

9.1.3. Linearizability

Now to the definition of linearizability: Typically the effect of a concrete data structure operation occurs before the operation actually returns. E.g., the increment operation `INC` from Figure 8.3 first increments the counter and then returns its previous value in a subsequent step. The definition of linearizability takes this into account according to the following modifications of a history H that can be performed before H is compared with abstract histories:

- For the running operations that have already taken effect but not yet returned, matching return events must be added to H. This gives a history $H + L_R$ with a list L_R of return events for pending invocations in H.
- The remaining pending invocations in $H + L_R$ represent running operations that have not taken effect yet. These are removed from $H + L_R$ using a function *complete*. The history $\mathit{complete}(H + L_R)$ only consists of matching pairs.

A legal history H is linearizable w.r.t. some sequential history H_s, if all events in H can be reordered to get H_s as follows:

- The reordering must preserve the sequential behavior of each process p in H, i.e., $complete(H + L_R) \downarrow_p = H_s \downarrow_p$.
- The reordering must preserve the order of executions in H that do not overlap in time, i.e., operation invocations that occur after a return event in H, must also occur after this return event in H_s.

We formalize these two reordering constraints as follows:

Definition 9.5 (Linearizable Histories)
A legal history H is linearizable w.r.t. a sequential history H_s if predicate $linearizable(H, H_s)$ holds

$$linearizable(H, H_s) \equiv \exists\ R.\ legal(H + L_R) \wedge lin(complete(H + L_R), H_s)$$

where predicate lin is defined as

$$\begin{aligned} &lin(H, H_s) \equiv \exists\ bijective\ f\colon [0, ..., \#\ H) \rightarrowtail [0, ..., \#\ H_s)\ such\ that \\ &\quad (\forall\ n < \#\ H.\ H(n) = H_s(f(n))) \\ &\wedge \forall\ m, n, m', n' < \#\ H.\ mp(m, n, H) \wedge mp(m', n', H) \wedge n < m' \rightarrow f(n) < f(m') \end{aligned}$$

The last conjunct above ensures that the order of non-overlapping executions represented as $mp(m, n, H)$ and $mp(m', n', H)$ with $n < m'$ is preserved.[1]

With these prerequisites, we can define linearizability of our concurrent system $\texttt{SPAWN}_{n,H}$ as follows.

Definition 9.6 (Linearizability)
The concrete system $\texttt{SPAWN}_{n,H}$ is linearizable if its histories H are linearizable w.r.t. some sequential history h_s of the abstract specification:

$$\begin{aligned} &Init_H(CS), \texttt{SPAWN}_{n,H}(CS), \Box\ (CS' = CS'' \wedge H' = H'') \\ \vdash\ &\Box\ ((\exists\ h_s.\ ahist(h_s) \wedge linearizable(H, h_s)) \wedge (\exists\ h_s.\ ahist(h_s) \wedge linearizable(H', h_s))) \end{aligned}$$

Definition 9.6 requires that for every prefix of H there is some sequential history (stored in a static variable) h_s of the abstract specification that serves as a linearizability witness.

[1] This constraint also ensures that matching pairs are preserved, i.e., the constraint $\forall\ m, n.\ mp(m, n, H) \rightarrow f(m) + 1 = f(n)$ holds implicitly.

9.2. Possibilities

Verifying an implementation linearizable by directly showing Definition 9.6 is tedious as we have illustrated in Section 8.3. Therefore, possibilities have been already introduced in [48] as a more practical proof method which rests on the intuition that a linearizable operation appears to take effect instantly between its invocation and return.

Based on this intuition, possibilities define admissible behaviors of linearizable algorithms as sequences of possibility steps: A possibility step is either an invocation step (*Invoke*), an abstract atomic operation step (*Linearize*), or a return step (*Return*). Technically, we recursively define possibilities $Poss(H, R, AS)$ as a sequence of possibility steps $Poss(H_0, R_0, AS_0, H, R, AS)$ that starts with an empty history H_0, an empty set R_0 of return events and an initial state AS_0.

$$Poss(H, R, AS) \equiv \exists\ AS_0.\ AInit(AS_0) \wedge Poss([\,], \emptyset, AS_0, H, R, AS)$$

Definition 9.7 (Sequences of Possibility Steps)
For concurrent histories H, H', sets R, R' of return events for pending invocations (in H/H') and abstract states AS, AS', a sequence of possibility steps consists of invocation, linearization or return steps

$$Poss(H, R, AS, H', R', AS') \equiv (Invoke \vee Linearize \vee Return)^*$$

where * denotes the reflexive and transitive closure here.

Invocation. In an invocation step, the executing process must not have a pending invocation event in H; the step changes neither the abstract state nor the return set.

$$\begin{aligned} &Invoke(H, R, AS, H', R', AS') \equiv \exists\ p, I, In. \\ &\quad H' = H + inv_p(I, In) \wedge nopi_p(H) \wedge R = R' \wedge AS = AS' \\ &\text{where } nopi_p(H) \equiv \forall\ n < \#\ H.\ pi(n, H) \rightarrow H(n).p \neq p \end{aligned}$$

Linearization. The execution of a linearization step corresponds to executing an abstract atomic transition AOP and adding a corresponding return event to R; the step does not change the history.

$$\begin{aligned} &Linearize(H, R, AS, H', R', AS') \equiv H = H' \wedge \\ &\quad \exists\ n, Out. \quad pi(n, H) \wedge AOP(H(n).i)(H(n).in, AS, AS', Out) \\ &\qquad\qquad \wedge R' = R + ret_{H(n).p}(H(n).i, Out) \wedge \forall\ e.\ e \in R \rightarrow e.p \neq H(n).p \end{aligned}$$

In the following, we write $Lin_{I,Out}$ for a linearization step of operation I with output Out.

Return. A return step completes a running operation that has already linearized by removing its return event e from R and adding it to the history.

$$\begin{aligned} &Return(H, R, AS, H', R', AS') \equiv \\ &\quad AS = AS' \wedge \exists\ e.\ e \in R \wedge H' = H + e \wedge R' = R \setminus \{e\} \end{aligned}$$

Based on these introduced definitions we can prove that possibilities are a proof method for linearizability:

Theorem 9.1 (Possibilities Imply Linearizability)
If every history prefix of the concurrent system $\texttt{SPAWN}_{n,H}$ *corresponds to some possibility* $Poss$

$$\begin{aligned} & Init_H(CS), \texttt{SPAWN}_{n,H}(CS), \Box\ H' = H'' \\ \vdash\ & \Box\ ((\exists\ r, as.\ Poss(H, r, as)) \wedge (\exists\ r, as.\ Poss(H', r, as))) \end{aligned} \tag{9.1}$$

then the concurrent system $\texttt{SPAWN}_{n,H}$ *is linearizable.*

The proof of this theorem follows directly from the following predicate logic lemma

$$Poss(H, R, AS) \rightarrow \exists\ H_s.\ linearizable(H, H_s) \wedge linval(H_s, AS)$$

which is not in the scope of this work, but can be found online [56].

9.3. Proof Method: RG Reasoning with Possibilities

This section introduces our compositional generic proof method for linearizability that combines possibilities and rely-guarantee reasoning. It is compositional, as it reduces the proof of the overall global property of linearizability to the verification of an RG assertion for partial correctness for an individual process. The basic idea is that each process must preserve possibilities as an additional guarantee condition for linearizability. This specific guarantee performs a step-local backward simulation, which we have transferred from the complete proof obligation [88] to our temporal logic setting. The soundness proof of our proof method has been mechanized in KIV. We do not prove a completeness result here, but we conjecture that it is complete for linearizability for the same reasons as in [88] (and the fact that RG reasoning is complete w.r.t. to the Owicki-Gries method [86] used in [88]).

Our proof method is a linearizability-specific instance of the parallel decomposition rule 6.1 as we explain in the following: The method requires to show a central proof obligation for an individual process p, similar to RG assertion (6) from rule 6.1

$$\begin{aligned} & nopi_p(H), Inv_H(CS), \Box\ Out' = Out'' \\ \vdash\ & [R_p(CS', H', CS'', H'') \wedge R_p^{lin}(H', H''), \\ & G_p(CS, H, CS', H') \wedge G^{lin}(CS, H, CS', H'), \\ & Inv_H(CS), \texttt{COP}_{p,H}(I, In; CS, Out)]\ true \end{aligned} \tag{9.2}$$

but where the system state S is renamed to CS and extended with a history variable H as explained before.

Moreover, we make the following adaptations: We model local states of all processes using a function $CS.LSf : nat \rightarrow lstate$, similar to Section 6.2, and the specification of

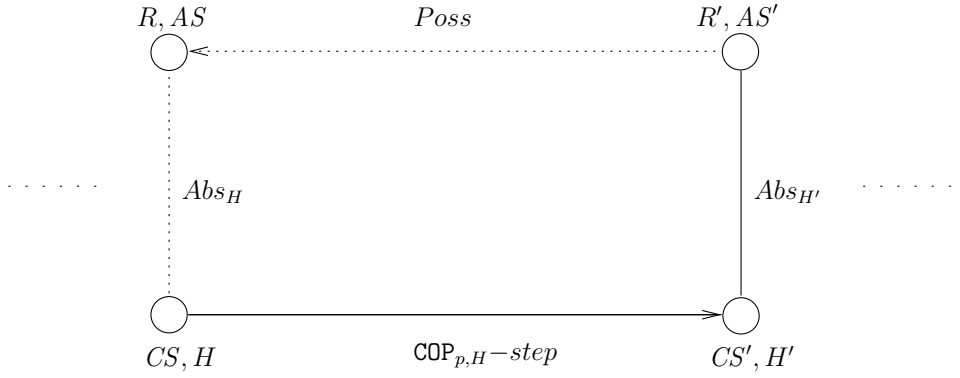


Figure 9.1.: Step-Local Backward Simulation for Linearizability.

$\texttt{COP}_{p,H}$

$$
\begin{array}{l}
\texttt{COP}_{p,H}(I, In; CS, Out)\ \{ \\
\quad H := H + inv_p(I, In),\ CS.LSf_p := init(I); \\
\quad \texttt{COP}_p(I, In; CS, Out); \\
\quad H := H + ret_p(I, Out) \\
\}
\end{array}
$$

now initializes (a part of) the local variables with the invocation transition based on a function $init\colon index \to lstate$. This is required, since directly using a **let** for the initialization can hide relevant local state from the specifications. For brevity, we leave some simple properties implicit in (9.2) such as the fact that each process p works exclusively on $CS.LSf_p$ and that the evolving concurrent history H is always legal.

The essential adaptation for linearizability in (9.2) is the following: We strengthen the individual guarantee G_p from premise (6) of rule 6.1 with an additional guarantee G^{lin} which ensures linearizability via step-local backward simulation as Figure 9.1 shows.

$$
\begin{array}{l}
G^{lin}(CS, H, CS', H') \equiv \forall\ R', AS'. \\
\qquad Abs_{H'}(CS', R', AS') \to \exists\ R, AS.\ Abs_H(CS, R, AS) \wedge Poss(H, R, AS, H', R', AS')
\end{array}
$$

That is, for each program transition of $\texttt{COP}_{p,H}$ that leads from a state CS, H to CS', H', we must show that each abstract state R', AS' that is related to CS', H' according to an abstraction relation $Abs_{H'}$, must have been reached by a finite sequence of possibility steps starting from some abstract state R, AS that Abs_H relates (backwards) to CS, H.

This step-local proof technique with possibilities requires to also strengthen the rely conditions R_p from rule 6.1 with an additional rely R_p^{lin} which ensures that after adding

an invoke event inv_p to H

$$R_p^{lin}(H', H'') \equiv \quad (nopi_p(H') \rightarrow nopi_p(H'')) \\ \wedge \forall\, n.\ pi(n, H') \wedge H'(n).p = p \rightarrow pi(n, H'') \wedge H'(n) = H''(n)$$

this event remains pending and unchanged in H throughout the entire execution of $\texttt{COP}_{p,H}$. Otherwise, propagating possibilities during p's execution would be impossible.

Moreover, we assume that the abstraction relation Abs_H is *total* over invariant states.

$$Inv_H(CS) \rightarrow \exists\, R, AS.\ Abs_H(CS, R, AS)$$

The remaining predicates of rule 6.1 are instantiated as follows: The pre-condition of a process p now states that p has no pending invocations in H ($nopi_p(H)$). The post-condition is trivial since data structure algorithms typically do not compute a result value for a final state. Finally, predicate $Init_H$ assumes that in the initial overall system state, history H is empty ($H = [\,]$) and the invariant holds. Finally, each initial concrete state corresponds to some abstract initial state ($AInit(AS)$) where no process has linearized yet ($R = \emptyset$).

$$Init_{[\,]}(CS) \wedge Abs_{[\,]}(CS, R, AS) \rightarrow AInit(AS) \wedge R = \emptyset \tag{9.3}$$

Theorem 9.2 (Soundness of the Generic Proof Method for Linearizability)

With the side conditions above on the used predicates, proof obligation (9.2) is a proof method for linearizability 9.6.

Proof Theorem 9.1 together with the following sublemma

$$\begin{array}{l} Init_H(CS), \texttt{SPAWN}_{n,H}(CS), \Box\ (CS' = CS'' \wedge H' = H'') \\ \vdash \Box\ (\quad (\forall\, r, as.\ Abs_H(CS, r, as) \rightarrow Poss(H, r, as)) \\ \qquad \wedge\ (\forall\, r, as.\ Abs_H(CS', r, as) \rightarrow Poss(H', r, as)))) \end{array} \tag{9.4}$$

ensure the correctness of our proof method.

Theorem 9.1 is applicable here since the invariant Inv_H (which implies Abs_H for some abstract state) holds in each state of the concurrent system $\texttt{SPAWN}_{n,H}$. This is implied by the following RG assertion

$$\begin{array}{l} Init_H(CS) \\ \vdash [R{\leq_n}(CS', H', CS'', H'') \wedge R^{lin}{\leq_n}(H', H'') \wedge G^{lin}(CS', H', CS'', H''), \\ \quad G{\leq_n}(CS, H, CS', H') \wedge G^{lin}(CS, H, CS', H'), Inv_H(CS), \texttt{SPAWN}_{n,H}(CS)]\ true \end{array} \tag{9.5}$$

that follows from rule 6.1 with the instances above. In (9.5) the global rely conditions $R^{lin}{\leq_n}$ result from lifting R_p^{lin} to the overall system state (see, e.g., the definition of $R{\leq_n}$ for rule 6.1). Note that we also strengthen the overall system rely to additionally sustain predicate G^{lin}. This can be easily justified since each process ensures G^{lin} in

its steps even with the weaker environment assumptions in (9.2). It is also easy to see that the predicate logic premises of rule 6.1 are satisfied for our instances above.

Thus it remains to prove Sublemma (9.4). Its proof first rewrites $\texttt{SPAWN}_{n,H}$ with the RG assertion (9.5) and $Init_H$ with the invariant Inv_H plus the essential property

$$\forall\, r, as.\ Abs_H(CS, r, as) \rightarrow Poss(H, r, as) \tag{9.6}$$

which holds initially according to (9.3). Then the proof proceeds by induction over the $\Box$ formula in the succedent, rule (2.5). After one step of symbolic execution (see Theorem 2.2), the proof goal is repeated and thus the induction hypothesis closes it. This is because the linearizability condition G^{lin} holds over both the last program and environment transition which propagates property (9.6) to the first primed state and the next unprimed state of the given interval. $\Box$

9.4. Case Study: A Wait-Free Multiset

This section illustrates the application of our generic proof method for linearizability to a challenging case study. It is a novel fixed-size multiset implementation with concurrent insert, lookup and delete operations for a given element x. Our multiset operations are wait-free, i.e., each operation terminates in a finite number of steps, independent of environment behavior. Wait-free implementations are particularly useful in real-time settings where the number of execution steps of an operation must be known beforehand.

While our multiset implementation is pretty simple, it has rather intricate correctness arguments: There are potential linearization points [24] that change the abstract representation and linearize several other running processes. The linearizability and wait-freedom proofs for our multiset have been mechanized in KIV [61].

Our work on the wait-free multiset started by looking at [29] where a *lock-based* multiset implementation *without* a delete operation is verified. (We consider their version in the next chapter where we illustrate our refinement-based proof method for linearizability.) In fact, we and the authors of [29] first thought that adding a delete operation to the implementation would violate linearizability. However, we found out later on that our presumed counter-example was flawed. Our linearizability result for our wait-free multiset suggests that adding a (blocking) delete operation to their implementation should also be possible. Thus it solves an open challenge according to [99].

9.4.1. The Wait-Free Multiset Operations

Figure 9.2 shows our simple multiset algorithms. The elements $x, y, \ldots$ of the multiset are stored in a shared array Ar of fixed size $N \neq 0$ where each array slot can either contain an element or be empty as indicated by the special value *empty*. Each operation gets an input element x and sequentially runs through the array Ar starting at index 0.

As soon as the insert operation `INSERT`(x) finds an empty slot, it atomically replaces *empty* with x using a CAS operation (line I2) and returns true. If no empty slot is found during the search, the insert operation returns false. Similarly, a delete operation `DELETE`(x) assigns *empty* to the first slot in Ar that contains x.[2] The lookup operation `LOOKUP`(x) does not use CAS, but only atomic reading. It succeeds and returns true if it finds the searched element x throughout its scan of Ar, otherwise it returns false.

It is simple to see that our implementation is wait-free, since each operation takes at most N steps to complete its sequential scan. However, it is not obvious to see that it is linearizable, as we discuss in Section 9.4.4.

[2] A possible variation of the `INSERT`/`DELETE` algorithms would be to only execute a CAS if its test has previously succeeded. Such performance considerations are, however, not in the scope of this work.

```
slot: elem | empty;
var Ar: array[0..N-1] of slot;

INSERT(x: elem): bool
  var i: int;
  for (i := 0; i < N; i++) {
    if CAS(Ar[i], x, empty) {
      return true
    }
  };
  return false

DELETE(x: elem): bool
  var i: int;
  for (i := 0; i < N; i++) {
    if CAS(Ar[i], empty, x) {
      return true
    }
  };
  return false

LOOKUP(x: elem): bool
  var i: int;
  for (i := 0; i < N; i++) {
    if (Ar[i] = x) {
      return true
    }
  };
  return false
```

Figure 9.2.: The Wait-Free Multiset Operations `INSERT`, `LOOKUP` and `DELETE`.

9.4.2. The Concrete Multiset Specification in RGITL

Figure 9.3 shows the formal specification of the concrete multiset algorithms in RGITL. It is close to the pseudo code from Figure 9.2 and uses the same array data structure; some minor syntactic differences result from instantiating the generic proof method and using our abstract programming language, respectively. In particular, the algorithm keeps the following *local* information in its tuple variable *Stat*: *Stat.op* is the operation status of the current process which can be either *ins*, *del* or *lkp* for the three operations, respectively; *Stat.in* is the input element that the operation searches for; *Stat.ix* is the current index position of the running operation and *Stat.found* is a boolean flag that determines whether the operation has found the searched element or not. When an operation returns, it assigns the value of *Stat.found* to the output parameter *Out*. To model the atomic CAS instructions in `INSERT` and `DELETE`, we use the **if**$*$ construct which executes its test and the following instruction in parallel. Thus some uncritical local transitions (e.g., the increment of *Stat.ix*) are also executed in parallel with the test. (We have also verified a version where these are executed in extra steps.)

In the following, we write *Op*, *In*, ... as abbreviations for *Stat.op*, *Stat.in*, etc. In the initial configuration the operation index *Op* is one of *ins*/*del*/*lkp*, the current position *Ix* is 0 and *Found* is false.

```
INSERT(Stat, Ar, Out) {
 while ¬ Stat.found ∧ Stat.ix < N do {
   if∗ Ar[Stat.ix] = empty then {
     Ar[Stat.ix] := Stat.in, Stat.found := true
   } else {
     Stat.ix := Stat.ix + 1
   }
 };
 Out := Stat.found}

DELETE(Stat, Ar, Out) {
 while ¬ Stat.found ∧ Stat.ix < N do {
   if∗ Ar[Stat.ix] = Stat.in then {
     Ar[Stat.ix] := empty, Stat.found := true
   } else {
     Stat.ix := Stat.ix + 1
   }
 };
 Out := Stat.found}

LOOKUP(Stat, Ar, Out) {
 while ¬ Stat.found ∧ Stat.ix < N do {
   if∗ Ar[Stat.ix] = Stat.in then {
     Stat.found := true
   } else {
     Stat.ix := Stat.ix + 1
   }
 };
 Out := Stat.found}
```

Figure 9.3.: Specification of Wait-Free `INSERT`, `DELETE` and `LOOKUP` Methods in RGITL.

9.4.3. The Abstract Semantics of the Multiset Operations

Linearizability (see Section 9.1) requires to specify the semantics of the multiset implementation in terms of an abstract specification with atomic operations. The abstract state of type $mset$ is an algebraic multiset of elements, which is initialized to be empty according to predicate $AInit(Ms)$ for the abstract state variable $Ms\colon mset$. The abstract atomic operations are defined as follows:

The abstract lookup relation $ALookUp$ leaves the multiset unchanged ($Ms' = Ms$) and atomically sets the boolean output value B to true iff its input element x occurs in the multiset at least once before the transition ($x \in Ms$).

$$ALookUp(x, Ms, Ms', B) \equiv Ms' = Ms \wedge (B \leftrightarrow x \in Ms)$$

The abstract relation $ADelete$ atomically removes one occurrence of its input element x from the multiset and returns true iff x is currently in Ms, otherwise it leaves the multiset unchanged and returns false.

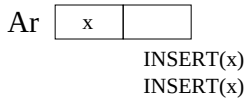
$$ADelete(x, Ms, Ms', B) \equiv Ms' = Ms \setminus \{\{x\}\} \wedge (B \leftrightarrow x \in Ms)$$

Finally, the insert relation $AInsert$ either adds x to the current multiset and returns true, or it nondeterministically returns false leaving the multiset unchanged

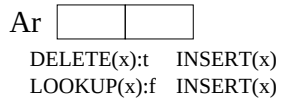
$$AInsert(x, Ms, Ms', B) \equiv Ms' = Ms \cup \{\{x\}\} \wedge B \vee Ms' = Ms \wedge \neg\, B$$

Note that we do not guarantee that the `INSERT` method only returns false if the multiset is full w.r.t. a predefined bound (typically the constant size of an underlying array). Introducing an abstract bound on the number of elements in the multiset would make the implementation not linearizable as the following concurrent execution for an array of size 2 shows:

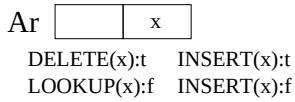
Assume $Ar[0] = x$ and $Ar[1] = empty$ initially. Two insert operations for element x are concurrently invoked. Since $Ar[0] = x$ both operations first proceed to the second slot, but without executing their CAS operations yet:



While these two insert operations are preempted, the first x gets concurrently deleted and then a lookup operation for x returns with false, since the array is empty after the previous deletion of x:



Next, the two preempted insert operations run to completion, one adding x to the array and returning true and the other one returning false, since it finds the slot at index 1 not being empty:

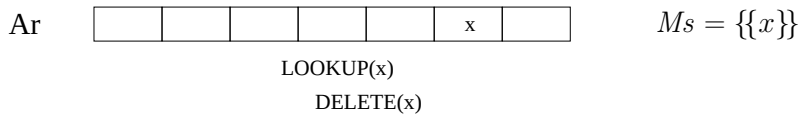


Now assume that the abstract multiset is initially $\{\{x\}\}$ and bounded to contain at most 2 elements. Moreover, let the abstract insert operation now only fail if the multiset is full. Then the concurrent history sketched above is not linearizable: In a corresponding sequential execution of the abstract multiset, the lookup that returns false would have to be after the succeeding delete operation, since the other two operations do not remove elements from the multiset. Moreover, the execution can not start with an insert that returns false, since there is still room for one element initially. It also can not start with the successful insert operation, because otherwise a lookup can not return false as there is still at least one element x in the multiset. Hence, the abstract execution must start with the delete operation, but then both insert operations would succeed. Thus introducing a bound on the number of elements in the multiset makes the implementation nonlinearizable.[3]

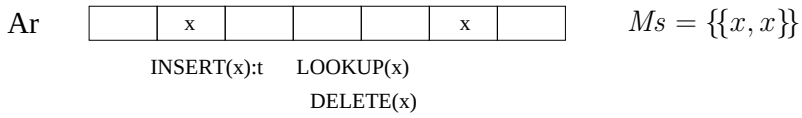
9.4.4. Challenges of Proving the Multiset Linearizable

Initially, we had thought that our implementation can not be linearizable due to an informal argument that later on turned out to be wrong. Consider the following concrete execution where Ms is the represented abstract state:

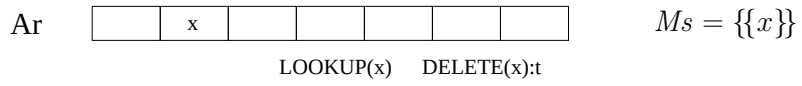
A lookup and a delete operation concurrently search for the same element x that lies ahead of their current positions but they have both not reached x's position yet:



Next, both operations are preempted and a concurrent insert operation successfully inserts x below the current search indices of lookup and delete:

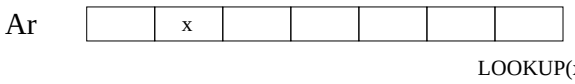


Then the delete operation runs to completion and removes x from the upper part of the array:



[3]This example also shows that an atomicity check [35] would fail for our multiset, since running the concrete code without interruption as an abstract specification does not offer the possibility to fail non-deterministically.

Finally, the lookup operation completes and returns false:



$Ms = \{\{x\}\}$

Intuitively, it seemed to us that this concurrent behavior is not linearizable, since the abstract lookup operation returns false although x is in the abstract multiset throughout its execution. However, according to the definition of linearizability (9.6), the abstract effect of the insert and delete operations may be reordered here, since their executions *do* overlap in time. Thus a linearization history exists where the delete operation takes effect before the insert. That is, the concurrent history

$$inv_p(lkp, x),\ inv_q(del, x),\ inv_r(ins, x),\ ret_r(ins, true),\ ret_q(del, true), ret_p(lkp, false)$$

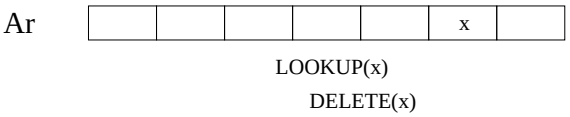
can be correctly linearized to the sequential history

$$inv_q(del, x),\ ret_q(del, true),\ inv_p(lkp, x),\ ret_p(lkp, false),\ inv_r(ins, x), ret_r(ins, true)$$

where first the delete operation takes effect, deleting the initial occurrence of x in the multiset, and thus making a lookup with false possible.

This concurrent execution already motivates a central idea of our linearizability proofs for the multiset: Successful delete operations must potentially take effect *early* during their execution, before they actually delete their element from the array. Intuitively, this ensures that we can "move" their abstract effect towards the time of their invocation. But then the abstract representation becomes a collection of multisets, since potentially linearizing a delete operation does not leave the abstract state unchanged. To better understand this, reconsider the previous execution:

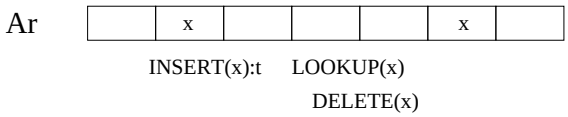
Initially, no process is running and the abstract representation is merely $\{\{\{x\}\}\}$. In general, when no running delete operation exists, the abstract representation is given by the elements in the array. As soon as a delete operation starts, it might have already linearized which gives the following representation



$Ms = \{\{\{\}\}, \{\{x\}\}\}$

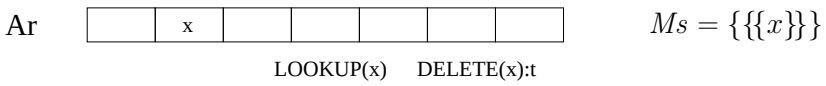
where the possible empty abstract multiset $\{\{\}\}$ results from deleting one occurrence of x from the initial multiset $\{\{x\}\}$.

After the insert operation succeeds, the abstract representation is either $\{\{x, x\}\}$ or, if the delete has potentially linearized, then one occurrence of x is removed which gives $\{\{x\}\}$ as another possible representation:



$Ms = \{\{\{x\}\}, \{\{x, x\}\}\}$

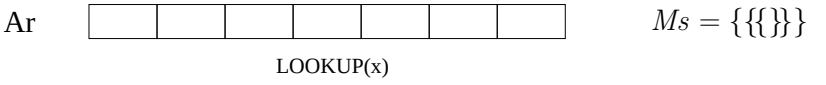
Finally, as soon as the delete operation terminates, $\{\{x\}\}$ becomes the unique abstract representation again:



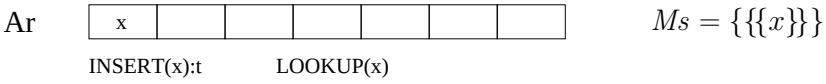
Note that in the execution above, the lookup operation must linearize to false *with* the potential linearization of the delete operation that removes the last occurrence of x from the multiset. If there were any delay between these two linearizations, then a concurrent `INSERT` might insert x below the current position of the lookup operation and linearizing to false would no longer be possible, since x would be contained in any possible abstract multiset. Thus the linearization point of the delete operation is also an *external* linearization point for all lookup/delete operations that can now complete with false.

Further challenges to prove the multiset linearizable exist for a lookup that returns false (and symmetrically for a delete that returns false). Consider the following execution where the array is initially empty:

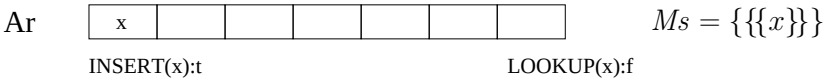
A lookup operation is preempted after it passes the first array-slot:



Then a concurrent insert occurs at the first slot



and lookup completes with false



Consequently, a lookup/delete that returns false must potentially linearize with false before passing the first slot of the array, since after passing the first slot, its searched element can be concurrently inserted at a lower position and linearizing to false becomes impossible.

Together, successful delete operations, plus lookup/delete operations that return false, must potentially take effect *early* during their execution.

9.4.5. The Abstraction Relation for the Multiset

Our previous considerations are formally taken into account by the instantiation of the abstraction relation Abs_H of our proof method (see Theorem 9.2) as we explain next: The abstraction relation Abs_H for the multiset relates a given concrete state

$(Statf, Ar)$[4] with concurrent history H, to an abstract state R, Ms where R is the set of return events for those running operations that have already passed their linearization point but not yet returned. Predicate Abs_H is recursively defined as a disjunction of 4 subformulas which we detail in the following.

$$Abs_H(Statf, Ar, R, Ms) \equiv BASE \vee DELt \vee DELf \vee LKPf$$

The base case is

$$BASE \;\equiv\; Ms = Absf(Ar) \wedge R = Linsf(Statf, Ar)$$

where the abstract multiset Ms consists of all elements in Ar that are computed by function $Absf$. Set R corresponds to precisely those running processes which have either not found their searched element and are at the end of their scan ($Ix_p = N$) or which have found it and set their found-flag to true. The return events of these processes are computed by function $Linsf$ as

$$e \in Linsf \leftrightarrow \exists\, p.\; e = ret_p(Op_p, Found_p) \wedge expi_p(n, H) \wedge (Ix_p < N \rightarrow Found_p)$$

where predicate $expi_p(n, H)$ states that p has a pending invoke in H at position n. (Our definition of Abs_H is thus total.)

The second disjunct in the definition of Abs_H describes the early linearization of a running delete operation that can potentially delete the searched element that lies ahead of its current position

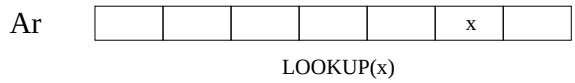
$$\begin{aligned} DELt \equiv \exists\, & p, n. \\ & Op_p = del \wedge \neg\, Found_p \wedge Ix_p \leq n \wedge n < \#Ar \wedge Ar[n] = In_p \\ \wedge\; & (\forall\, n_0.\; Ix_p \leq n_0 < n \rightarrow Ar[n_0] \neq In_p) \wedge ret_p(del, true) \in R \\ \wedge\; & Abs_{H+ret_p(del,true)}(Statf, Ar[n]{:=}empty, R \setminus \{ret_p(del, true)\}, Ms) \end{aligned}$$

The recursive call $Abs_{H+ret_p(del,true)}$ is computed for the state $Statf, Ar[n]{:=}empty$ where the delete process p has removed the first occurrence of its input element In_p at array position n (written $Ar[n]{:=}empty$) and then run to completion, removing $ret_p(del, true)$ from R and adding it to history H according to the *Return* possibility in Section 9.2.

As an aside, $DELt$ removes the *first* occurrence of the searched element that lies ahead of the current process' position. If the algorithm were to delete some random occurrence ahead (or would scan the array in reverse order from its upper bound to its lower bound), then it would no longer be linearizable. To better understand why, consider the following concurrent execution (starting with $Ms = \{\{x\}\}$):

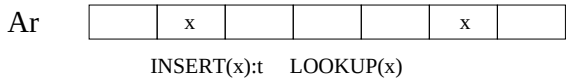
First, a process p invokes a lookup operation for an element x that is ahead of its current position:

[4] The higher-order variable $Statf\colon nat \rightarrow status$ lifts the status variable $Stat$ of each process to a global setting with an arbitrary finite number of processes. We typically write Op_p, $In_p, \ldots$ as abbreviations for $Statf_p.op$, $Statf_p.in, \ldots$



$Ms = \{\{\{x\}\}\}$

Next, process p is preempted and another process q concurrently inserts x below the current position of lookup:



$Ms = \{\{\{x, x\}\}\}$

If a delete operation would now delete the occurrence of x in the upper part of Ar, then the running lookup operation could return with false which would violate linearizability: Remember that the definition of linearizability (9.6) would require that the insert must precede the delete in a corresponding abstract execution, since their execution times would *not* overlap here. But then it is simple to see that the lookup that returns false can not occur at any position of a corresponding abstract history.

Now to the third disjunct in the definition of Abs_H. It considers a running delete process that potentially linearizes to false as it does not see its searched element ahead

$$\begin{aligned} DELf \equiv \exists\ p.\ & \\ & Op_p = del \wedge \neg\ Found_p \wedge Ix_p < \#Ar \\ \wedge\ & (\forall\ n.\ Ix_p \leq n < \#Ar \rightarrow Ar[n] \neq In_p) \wedge ret_p(del, false) \in R \\ \wedge\ & Abs_{H+ret_p(del,false)}(Statf, Ar, R \setminus \{ret_p(del, false)\}, Ms) \end{aligned}$$

The last disjunct for a lookup that returns false is symmetric to $DELf$

$$\begin{aligned} LKPf \equiv \exists\ p.\ & \\ & Op_p = lkp \wedge \neg\ Found_p \wedge Ix_p < \#Ar \\ \wedge\ & (\forall\ n.\ Ix_p \leq n < \#Ar \rightarrow Ar[n] \neq In_p) \wedge ret_p(lkp, false) \in R \\ \wedge\ & Abs_{H+ret_p(lkp,false)}(Statf, Ar, R \setminus \{ret_p(lkp, false)\}, Ms) \end{aligned}$$

It is easy to see that the recursion in Abs_H is well-founded: The proof is by induction over the number of running processes which decreases in the recursive calls where the running process p returns.

The idea to define the possible abstract multiset representations by executing running operations to completion is taken from [88], which mechanizes the linearizability proof for the intricate HW queue algorithm [48]. The proofs for the HW queue can be similarly done with our generic proof method here. They are slightly more challenging than those for the multiset, as they must additionally consider the FIFO semantics of the underlying data structure.

9.4.6. Instantiating the Generic Proof Method

This section sketches the instances for the parameters of Theorem 9.2: The abstract state corresponds to Ms and the abstract operations AOP_p are those from Section

9.4.3. The instance for the concrete state variable CS is the tuple $Statf, Ar$ and the concrete operations $\texttt{COP}_p$ correspond to those from Figure 9.3: Depending on the operation index, $\texttt{COP}_{p,H}$ is thus one of $\texttt{DELETE}_p$, $\texttt{INSERT}_p$ or $\texttt{LOOKUP}_p$ with a preceding invocation/initialization step and a subsequent return step, respectively.

The remaining RG predicates have the following simple instances: In the overall initial system state the array is empty according to

$$Init \equiv \forall\ n < \#Ar.\ Ar[n] = empty$$

Hence $Init_H$ becomes $Init \wedge H = [\,]$.

The invariant merely states that the pending invocation of running processes corresponds to their status.

$$Inv_H \equiv \forall\ p.\ expi_p(n, H) \rightarrow H[n].in = In_p \wedge H[n].i = Op_p$$

The rely conditions R_p state that the length of the array is not concurrently changed (locality of $Statf_p$ is already implicitly ensured by the proof method)

$$R_p \equiv \#Ar' = \#Ar''$$

and the guarantee G_p is canonically defined as the rely conditions of all other processes

$$G_p \equiv \forall\ q \neq p.\ R_q$$

9.4.7. The Correctness Proof

The proof of linearizability requires to prove the premises of Theorem 9.2 with the instances above. The predicate logic premises are simple to prove. Thus we only focus on the central premise (6) in the following: It requires a temporal logic proof of an RG assertion for partial correctness for each individual multiset operation. We prove stronger versions for total correctness by induction over the variant $\#Ar - Ix_p$. Thus we show termination (wait-freedom) of each individual multiset operation at no additional cost (the symbolic execution of both types of RG assertions uses the same rules, see Section 4.3). These RG proofs lead to three central predicate logic lemmas where the step-local simulation relation G^{lin} must be propagated backwards over critical transitions, respectively:

The Invocation of `DELETE`. The most complex case results from the invocation of $\texttt{DELETE}_p(x)$ by some process p. (All other steps of `DELETE` have simple linearization points.)

Lemma 1 (The Invocation of $\texttt{DELETE}_p(x)$ Propagates G^{lin})

$$nopi_p(H), Inv_H(Statf, Ar), Abs_{H+inv_p(del,x)}(Statf_p{:=}(del, x, 0, false), Ar, R', Ms')$$
$$\vdash \exists\ R, Ms.\ Abs_H(Statf, Ar, R, Ms) \wedge Poss(H, R, Ms, H + inv_p(del, x), R', Ms')$$

Intuitively, the proof of Lemma 1 discerns two central cases: Either the invocation of $\texttt{DELETE}_p(x)$ considers an abstract multiset where x occurs several times. Then the

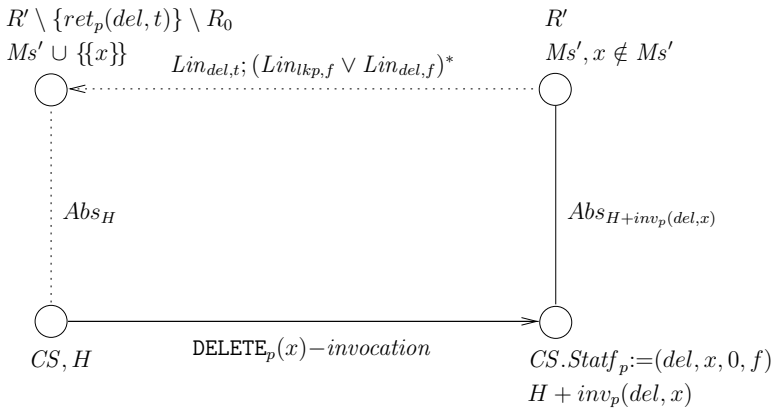


Figure 9.4.: Invocation of $\texttt{DELETE}_p(x)$ where x does not occur in the considered multiset Ms' after this step.

invocation step only (potentially) linearizes the current delete operation with true ($Lin_{del,t}$). Figure 9.4 shows the more difficult case where the abstract multiset before the invocation contains x only once, i.e., the represented multiset is $Ms' \cup \{\{x\}\}$ where $x \notin Ms'$. Then the invocation transition potentially linearizes the current delete process with true, plus a sequence $(Lin_{lkp,f} \vee Lin_{del,f})^*$ of running lookup/delete operations that can now return false (f). (The return events of these running lookup/delete processes are represented by the set R_0.)

Proof Technically, the proof of Lemma 1 is by well-founded induction over the number of running processes. Unfolding the definition of Abs_H in the antecedent gives four cases according to its four disjuncts. The most complex one is *DELt* where some running delete process completes and returns true. Two subcases must then be discerned, depending on whether the process that runs to completion is i) a second delete process q which is already running or ii) it is the current process p.

i) We prove the first case by applying the induction hypothesis to the concrete state where p has not been invoked yet but the other process q has already executed its successful delete operation to completion, setting the array slot at some position n to *empty*. The induction hypothesis is applicable for this state, since the number of running processes decreases by one when q returns. The other preconditions of the induction hypothesis follow with simple commutativity arguments for the completion of q and the invocation of p.

The result of applying induction is an abstract state with a set of linearized processes $\tilde{R}$ and the multiset $\tilde{Ms}$. Then we can simply use $\tilde{R} \cup \{ret_q(del, true)\}$ and $\tilde{Ms}$ as abstract representation for the existential quantifier in the succedent. It remains to prove the succedent formulas Abs_H and *Poss* for this instantiation. The abstraction relation can be easily shown by unfolding its definition and instantiating the quantifier in *DELt* with q and n. The possibilities predicate follows from two simple lemmas about adding invocation events to the history and swapping invoke and return events, respectively.

ii) The more complex case in which the current process p runs to completion has two subcases: 1) There are still occurrences of the deleted element x in the abstract multiset Ms' after deletion or 2) process p has just deleted the last occurrence and x is no longer in Ms'. In both cases induction can not be applied, since the number of running processes does not decrease. Therefore, we consider each case in an extra lemma.

1) The simpler case where x still occurs in Ms' after deletion follows with the following sublemma:

$$\begin{aligned} & Op_p = del, \neg\, Found_p, Ix_p < \#Ar, Ar[Ix_p] = x, \forall\, n < Ix_p.\ Ar[n] \neq x, \\ & Inv_H(Statf, Ar), Abs_H(.found_p{:=}true, Ar[Ix_p]{:=}empty, R', Ms'), x \in Ms' \\ \vdash\ & Abs_H(Statf, Ar, R' \setminus \{ret_p(del, true)\}, Ms' \cup \{\{x\}\}) \\ & \wedge\ Poss(H, R' \setminus \{ret_p(del, true)\}, Ms' \cup x, H, R', Ms') \end{aligned} \tag{9.7}$$

The lemma states that if some x still occurs in Ms' after removing x from the array at position Ix_p, then this transition linearizes exactly process p, i.e., the abstract representation before the transition is $R' \setminus \{ret_p(del, true)\}, Ms' \cup \{\{x\}\}$ and R', Ms' afterwards. (Remember that the linearization step *Linearize* in predicate *Poss* adds a corresponding return event to the set R.)

The proof of Sublemma (9.7) uses induction on the number of running processes and unfolds the definition of Abs_H in the antecedent. The most critical case here is *LKPf* (and symmetrically *DELf*), since we must ensure that p does not remove the last occurrence of x that lies ahead of a lookup process q in Ar. Otherwise, process q might return with false but linearizing any lookup to false is forbidden now as x occurs in the multiset. However, from the precondition $x \in Ms'$ plus the property $\forall\, n < Ix_p.\ Ar[n] \neq x$ which is given since p has just been invoked, we can easily deduce that q must still have an x ahead and can thus not return false now. (This is the formal equivalent to our previous informal explanation why a delete operation may not delete elements at random array positions, see Section 9.4.5.)

2) The more difficult case where x does no longer occur in Ms' after deletion follows with the following sublemma:

$$\begin{aligned} & Op_p = del, \neg\, Found_p, Ix_p < \#Ar, Ar[Ix_p] = x, Inv_H(Statf, Ar), \\ & Abs_H(.found_p{:=}true, Ar[Ix_p]{:=}empty, R', Ms'), x \notin Ms' \\ \vdash\ & \exists\, R_0 \subseteq R'.\quad (\forall\, e \in R_0.\ e.p \neq p \wedge e.i \neq ins \wedge \neg\, e.out) \\ & \qquad \wedge\ Abs_H(Statf, Ar, R' \setminus \{ret_p(del, true)\} \setminus R_0, Ms' \cup \{\{x\}\}) \\ & \qquad \wedge\ Poss(H, R' \setminus \{ret_p(del, true)\} \setminus R_0, Ms' \cup x, H, R', Ms') \end{aligned} \tag{9.8}$$

That is, if process p removes x at position Ix_p from the array and x does not occur in the corresponding multiset Ms', then this transition not only linearizes process p: Additionally, all running lookup and delete processes are now linearized with false if they do not have x ahead of their current position anymore after deletion. Technically, these additional processes are represented by R_0 in the succedent of

the sublemma. Thus the abstract representation before the deletion of the last x is $R' \setminus \{ret_p(del, true)\} \setminus R_0, Ms' \cup \{\{x\}\}$ and R', Ms' afterwards.

The proof of Lemma (9.8) is by induction over the number of running processes. Then the induction hypothesis is applied on the state where the process that is given by the definition of Abs_H in the antecedent, runs to completion. The rest of the proof applies similar arguments as in Sublemma (9.7). The main difference is that the subcases for *LKPf* (and symmetrically *DELf*) now add a lookup/delete process to the set $\tilde{R}$ that is given by the induction hypothesis.

This concludes the central case for *DELt* in the proof of Lemma 1. The remaining cases basically follow with the definition of Abs_H, applying induction or simple commutativity arguments. □

The Successful `LOOKUP` Transition. The critical transition of $\texttt{LOOKUP}_p(x)$ is when it finds the searched element x and sets its found flag to true. This case is covered by the following lemma. (All other steps of `LOOKUP` have simple linearization points.)

Lemma 2 (The Successful $\texttt{LOOKUP}_p(x)$ Propagates G^{lin})

$$
\begin{array}{ll}
 & Op_p = lkp, \neg\ Found_p, Ix_p < \#Ar, Ar[Ix_p] = x, Inv_H(Statf, Ar), \\
 & Abs_H(.found_p{:=}true, Ar, R', Ms') \\
\vdash & \quad Abs_H(Statf, Ar, R' \setminus \{ret_p(lkp, true)\}, Ms') \\
 & \quad \wedge\ Poss(H, R' \setminus \{ret_p(lkp, true)\}, Ms', H, R', Ms'), \\
 & \exists\ q, R_0. \quad Op_q = del \wedge \neg\ Found_q \wedge Ix_q < \#Ar \\
 & \qquad \wedge\ R_0 \subseteq R' \wedge (\forall\ e \in R_0.\ e.p \neq p \wedge e.i \neq ins \wedge \neg\ e.out) \\
 & \qquad \wedge\ Abs_H(Statf, Ar, R' \setminus \{ret_q(del, true), ret_p(lkp, true)\} \setminus R_0, Ms' \cup \{\{x\}\}) \\
 & \qquad \wedge\ Poss(H, R' \setminus \{ret_q(del, true), ret_p(lkp, true)\} \setminus R_0, Ms' \cup x, H, R', Ms')
\end{array}
$$

The first formula in the succedent of Lemma 2 corresponds to the simple case where x is in the considered multiset Ms' after the step and the successful lookup transition just linearizes the currently running process p with true. The second succedent formula, however, is necessary for the case where the considered multiset Ms' after the step does not contain x as shown in Figure 9.5. This situation is possible indeed, since there can be a running delete process q that (potentially) removes the last occurrence of x right *after* lookup process p linearizes with true. As a consequence, the step linearizes the current lookup process with true and the running delete process q with true. Furthermore, as in Lemma 1, a sequence $(Lin_{lkp,f} \vee Lin_{del,f})^*$ of currently running lookup/delete processes (represented by R_0) that can now return with false is linearized.

The proof of Lemma 2 is by induction over the number of running processes. It unfolds the definition of Abs_H in the antecedent which gives four cases. The critical case *DELt* where a process q deletes the last occurrence of x is closed by simply applying Sublemma (9.8) which gives the right abstract representation before the step. The other cases are covered by the first formula in the antecedent where only process p linearizes with true.

The Successful `INSERT` Transition. The critical transition of $\texttt{INSERT}_p(x)$ is when it inserts x in an empty array slot. The case corresponds to the following lemma. (All other steps have simple linearization points.)

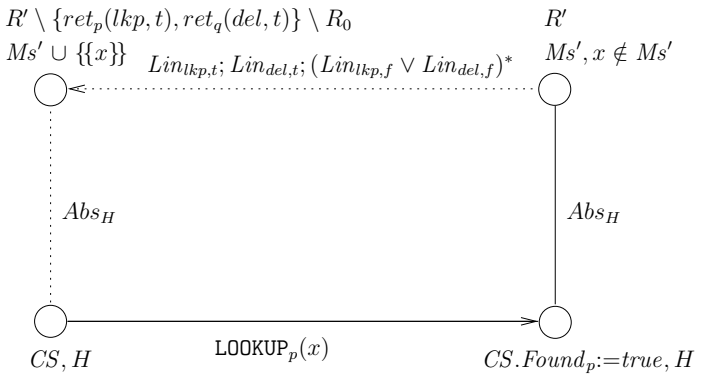


Figure 9.5.: Successful $\texttt{LOOKUP}(x)$ where x does not occur in the considered multiset Ms' after the step.

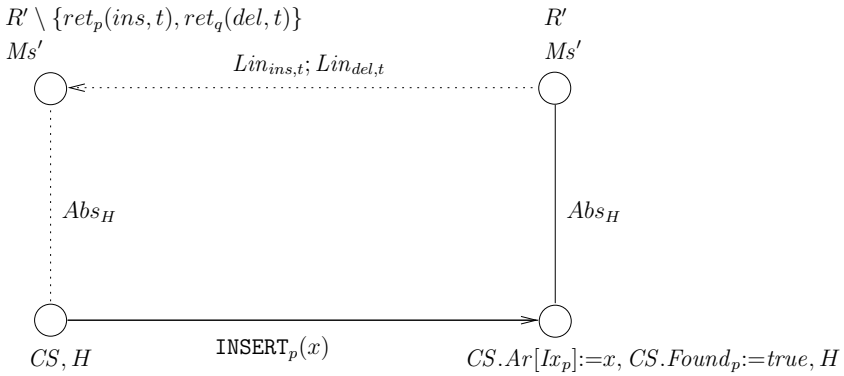


Figure 9.6.: Successful $\texttt{INSERT}(x)$ where x has been instantly removed from the representation by a running delete operation.

Lemma 3 (The Successful $\texttt{INSERT}_p(x)$ Propagates G^{lin})

$$
\begin{aligned}
& Op_p = ins, \neg\ Found_p, Ix_p < \#Ar, Ar[Ix_p] = empty, Inv_H(Statf, Ar), \\
& Abs_H(Statf_p.found{:=}true, Ar[Ix_p]{:=}x, R', Ms') \\
\vdash\ & Abs_H(Statf, Ar, R' \setminus \{ret_p(ins, true)\}, Ms' \setminus \{\{x\}\}) \\
& \wedge\ Poss(H, R' \setminus \{ret_p(ins, true)\}, Ms' \setminus \{\{x\}\}, H, R', Ms'), \\
& \exists\ q.\quad Op_q = del \wedge \neg\ Found_q \wedge Ix_q < \#Ar \\
& \qquad \wedge\ Abs_H(Statf, Ar, R' \setminus \{ret_p(ins, true), ret_q(del, true)\}, Ms') \\
& \qquad \wedge\ Poss(H, R' \setminus \{ret_p(ins, true), ret_q(del, true)\}, Ms', H, R', Ms')
\end{aligned}
$$

Lemma 3 states that a successful insertion either linearizes the insert process only (according to the first formula in the succedent), or it additionally linearizes a running delete operation that now potentially deletes the element that has just been inserted (according to the second formula in the succedent). In the latter case, predicate *Poss* corresponds to $Lin_{ins,t}; Lin_{del,t}$ as shown in Figure 9.6, and the abstract effects of the insert and delete operations are mutually canceled. Thus the abstract state before

the step is $R' \setminus \{ret_p(ins, true), ret_q(del, true)\}, Ms'$ and after the step it is R', Ms', respectively. Note that since the abstract multiset is not changed, no lookup/delete processes must be additionally linearized here with false as a consequence of the delete.

This linearization effect is typical for data structures that use elimination [46, 72]. Such algorithms are also provable with our generic proof method. The proof of Lemma 3 applies similar arguments as the proofs of the previous lemmas and is therefore omitted.

9.5. Summary

In this chapter we have introduced a generic proof method for linearizability that combines the well-known verification technique of identifying linearization points with RG reasoning, and we have mechanically proved its correctness w.r.t. the original definition [48]. Essentially, our work integrates the step-local backward simulation relation from [88] into our temporal logic setting with rely-guarantee reasoning. For a comparison with [88] and further related work see Chapter 13.

Furthermore, we have illustrated a challenging application of our proof method to verify a novel wait-free multiset implementation with intricate linearization points correct. The case study includes potential external linearization points where one instruction of a process can linearize several other running processes. Thus we have solved an open challenge according to [99]. The linearizability and wait-freedom proofs for the multiset are mechanized in KIV [61].

In the next chapter, we introduce a proof method for a restricted class of linearizable algorithms that makes several simplifications.

10. A Proof Method for Linearizability using Refinement

This chapter introduces a compositional proof method for linearizability that is based on a temporal logic refinement for an individual process. It is derived from the generic method from the previous chapter, and it is intended for the important subclass of linearizable algorithms where the linearization point of an operation corresponds to one of its own instructions. Section 10.1 defines the method and Section 10.2 illustrates its application to a non-trivial concurrent multiset implementation that uses fine-grained locking. The basic ideas of Section 10.1 correspond to our journal article [7], for a more detailed comparison see Chapter 13. The presentation in Section 10.2 follows our journal article [99].

10.1. Proof Method: RG Reasoning with Refinement

The application of our generic proof method for linearizability from Section 9.3 requires step-local backward reasoning with possibilities. The resulting technical overhead is acceptable for intricate examples where a step of one process can linearize several other processes and where potential linearization points may change the abstract state. However, for algorithms with less complex concurrent behaviors, a simpler proof method that avoids this overhead is desirable. This section introduces such a proof method. It is applicable to the important subclass of linearizable algorithms that meet the following two restrictions:

- Each running operation linearizes with one of its own steps (internal linearization point).
- Potential linearization points that depend on the future system execution do not change the abstract state.

Our proof method completely avoids backward reasoning with possibilities and uses RG reasoning and temporal logic refinement instead.

10.1.1. The Refinement-Based Proof Obligation for Linearizability

Similar to the generic proof method introduced in Section 9.3, the derived proof method also uses RG reasoning at its base to establish relevant safety formulas for the linearizability (refinement) proof. Different from the generic proof method, however, the RG conditions do not have a history variable as a parameter, but only refer to the system

state CS, see below. The underlying system model is now $\texttt{SPAWN}_{n,H}$ as in Section 9.3 but where the invocation transition of $\texttt{COP}_{p,H}$ does not initialize any local variables. (Thus the invocation and return trivially preserve our RG predicates which do not mention H.) Instead, we now use a predicate $Idle_p\colon cstate \to bool$ both as pre- *and* post-condition of each operation $\texttt{COP}_p$.

The proof method consists of two parts: The first part requires to establish suitable rely and invariant properties for an individual operation $\texttt{COP}_p$. The second part uses these properties in an additional temporal refinement proof that ensures linearizability of $\texttt{COP}_p$. Hence, the proof method requires to prove two central proof obligations:

The first proof obligation is the following RG assertion for partial correctness

$$\begin{aligned} & Idle_p(CS), Inv(CS), \Box\ Out' = Out'' \\ \vdash\ & [R_p(CS', CS''), G_p(CS, CS'), Inv(CS), \texttt{COP}_p(I, In; CS, Out)]\ Idle_p(CS) \end{aligned} \tag{10.1}$$

Proof obligation (10.1) ensures that from the local view of one process p, all environment transitions satisfy its rely R_p at all times and the invariant holds in each state. Hence, the temporal logic formula

$$RI \equiv R_p(CS', CS'') \wedge Inv(CS) \wedge Inv(CS')$$

holds locally at all times.

The second central proof obligation requires a refinement proof which identifies the linearization point of each $\texttt{COP}_p$. This refinement proof is carried out under the assumption of $\Box\ RI$.

$$\begin{aligned} & Idle_p(CS), \texttt{COP}_p(I, In; CS, Out), \Box\ RI, \Box\ Out' = Out'' \\ \vdash\ & \exists\ AS.\ \texttt{AOP}_p(I, In; AS, Out) \wedge \Box\ (Abs(CS) = AS \wedge Abs(CS') = AS') \end{aligned} \tag{10.2}$$

Note that (10.2) enforces that throughout the entire execution of $\texttt{COP}_p$, the abstract state AS must be equal to the value $Abs(CS)$ that is computed by the abstraction function Abs. Procedure $\texttt{AOP}_p$ from the succedent of (10.2) is specific for linearizability. It takes the same input In as the concrete program and must return the same output value Out upon termination:

$$\begin{array}{l} \texttt{AOP}_p(I, In; AS, Out)\ \{ \\ \quad \textbf{let}\ LOut\ \textbf{in}\ \{ \\ \quad\quad \textbf{bskip}^*; \\ \quad\quad \{AOP(I)(In, AS, AS', LOut') \wedge Out = Out' \wedge \neg\ \textbf{blocked} \wedge \textbf{step}\}; \\ \quad\quad \textbf{bskip}^*; \\ \quad\quad Out := LOut \\ \quad \}\ \} \end{array} \tag{10.3}$$

First $\texttt{AOP}_p$ executes arbitrarily many stutter steps $\textbf{bskip}^*$,[1] then it executes the atomic linearization transition $AOP(I)$. (The algebraic relation $AOP(I)$ is just as in the

[1] For non-blocking algorithms stutter steps can be simply modeled as **skip**, but for algorithms with locks, stuttering must also allow blocked steps that a process executes while waiting for a lock. Hence, $\textbf{bskip} \equiv \textbf{skip} \vee (\textbf{blocked} \wedge \textbf{step})$.

generic proof method.) After further stutter steps it typically returns with output value *Out*. (It might also stutter infinitely often if the concrete algorithm runs forever.)

Finally, similar to the generic proof method, we assume that the (partial) abstraction function *Abs* is *total* on states where predicate *Inv* holds

$$Inv(CS) \rightarrow \exists\ AS.\ Abs(CS) = AS$$

and that concrete initial states have a corresponding abstract initial state

$$Init(CS) \wedge Abs(CS) = AS \rightarrow AInit(AS) \tag{10.4}$$

Theorem 10.1 (Soundness of the Refinement-Based Proof Method) *If premises (1) - (4) and (7) of rule 6.1 hold with the predicates introduced above, then the RG proof obligation* (10.1) *and the refinement proof obligation (10.2) ensure that the concurrent system* $\texttt{SPAWN}_{n,H}$ *is linearizable.*

Roughly speaking, the soundness proof of Theorem 10.1 essentially shows that an abstract operation that executes $\mathbf{bskip}^*; AOP; \mathbf{bskip}^*$ preserves possibilities *Poss* (Definition 9.7) in its steps. (This is not difficult to see: Stutter steps trivially preserve possibilities and the atomic transition *AOP* corresponds to a linearization step *Linearize*.) Therefore, an abstract concurrent system ASPAWN that interleaves such abstract operations is linearizable (Theorem 9.1). Since each concrete operation refines such an abstract operation according to (10.2), an overall concurrent system CSPAWN that interleaves such concrete operations, refines ASPAWN and is therefore linearizable.

Proof The proof of Theorem 10.1 consists of two parts: i) We show that an abstract concurrent system $\texttt{ASPAWN}_{n,H}$ is linearizable (its implementation is given below). This proof uses an auxiliary variable *Statf* as a local program label for the linearizability status of each process. ii) We prove that the concrete system $\texttt{SPAWN}_{n,H}$ refines the abstract system $\texttt{ASPAWN}_{n,H,-Statf}$ where the status variable is removed (thus no initialization occurs with the invocation). Together, we show that

$$\texttt{SPAWN}_{n,H} \rightarrow \texttt{ASPAWN}_{n,H,-Statf} \rightarrow \texttt{ASPAWN}_{n,H}$$

where implications denote refinement (= trace inclusion) in RGITL. The left implication above corresponds to ii) and the right implication holds since the status function is an auxiliary variable for $\texttt{ASPAWN}_{n,H}$.[2] Hence, each execution of the concurrent system $\texttt{SPAWN}_{n,H}$ is also one of $\texttt{ASPAWN}_{n,H}$ which are all linearizable according to i). This concludes the main proof and it remains to prove i) and ii) in the following:

i) We prove that $\texttt{ASPAWN}_{n,H}$ is linearizable by instantiating our generic proof method for linearizability from Section 9.3. Thus the implementation of $\texttt{ASPAWN}_{n,H}$ corresponds to $\texttt{SPAWN}_{n,H}$ where the system state is the tuple $AS, Statf$.

[2] Currently, extending a program $\texttt{PROC}(S)$ with auxiliary state *Aux* such that the following refinement property $\texttt{PROC}(S) \rightarrow \exists\ Aux.\ init(Aux) \wedge \texttt{PROC}(S, Aux) \wedge \Box\ Aux' = Aux''$ holds, is not supported by a syntactic rule of RGITL. Therefore, we only assume semantically here that $\texttt{ASPAWN}_{n,H,-Statf}$ refines $\texttt{ASPAWN}_{n,H}$, since *Statf* is just such an auxiliary program label.

The status function $Statf : nat \to beforelin \mid (afterlin, output)$ encodes the local linearization status of each process as either being *beforelin* or *afterlin* with a specific output value. With the invocation of an operation, the status is initialized to *beforelin* and the linearization transition in $\texttt{AOP}_p$ changes the status to $(afterlin, LOut')$. Thus we use the following instance $\texttt{AOP}_{p,H}$ for $\texttt{COP}_{p,H}$ from the generic proof method

$$\begin{array}{l}\texttt{AOP}_{p,H}(Ax, In; AS, Statf, Out)\ \{\\ \quad H := H + inv_p(I, In), Statf_p := beforelin;\\ \quad \texttt{AOP}_p(I, In; AS, Statf, Out);\\ \quad H := H + ret_p(I, Out)\\ \}\end{array}$$

where the linearization transition in $\texttt{AOP}_p$ is now

$$AOP(I)(In, AS, AS', LOut') \wedge Statf'_p{:=}(afterlin, LOut') \wedge \ \ldots$$

Storing the local output $LOut'$ as part of the status ensures that the abstract output value of the linearization coincides with the output value Out that is written later on to the history H. This is required, since the introduced local variable $LOut$ is not visible outside of its scope.

The instantiation of most remaining parameters of the generic method is straightforward, e.g., the rely predicate is trivial, the invariant is just $Abs_H(AS, Statf, R, AS_0)$ for some R, AS_0 to ensure totality and so forth. The only remaining non-trivial instance is the abstraction relation

$$\begin{array}{l}Abs_H(AS, Statf, R, AS_0) \equiv AS = AS_0 \wedge \forall\ p.\\ \quad (nopi_p(H) \vee expi_p(n, H) \wedge Statf_p = beforelin \to R\downarrow_p = \emptyset)\\ \wedge\ (expi_p(n, H) \wedge Statf_p.st = afterlin \to ret_p(H(n).i, Statf_p.out) \in R\downarrow_p)\end{array}$$

which states that each running process p linearizes precisely when it executes the linearization transition in $\texttt{AOP}_p$, where R is the set of return events for pending invocations in H.

With these instances, the predicate logic premises of the generic proof method (9.2) are easy to prove. It remains to show its central proof obligation which is an RG assertion for partial correctness of $\texttt{AOP}_{p,H}$. The proof uses symbolic execution (see Theorem 2.2) and safety induction (4.4) for the RG assertion. In particular, the linearization transition follows with the following predicate logic sublemma:

$$\begin{array}{l}Statf_p = beforelin, expi_p(n, H), H(n) = inv_p(I, \_), Inv_H(AS),\\ AOP(I)(In, AS, AS', LOut')\\ \vdash G^{lin}(AS, Statf, H, AS', Statf'_p{:=}(afterlin, LOut'), H)\end{array}$$

The proof of this sublemma unfolds the definition of G^{lin} and instantiates the existentially quantified variables with $R \setminus \{ret_p(I, LOut')\}$, AS which is the abstract representation of the state before the linearization. This concludes the proof of i).

ii) It remains to prove that $\texttt{SPAWN}_{n,H}$ (where the initialization step is empty) refines $\texttt{ASPAWN}_{n,H,-Statf}$, i.e.,

$$\begin{aligned} &Init_H(CS), \texttt{SPAWN}_{n,H}(CS), \Box\ (CS' = CS'' \wedge H' = H'') \\ \vdash\ &\exists\ AS.\ AInit_H(AS) \wedge \texttt{ASPAWN}_{n,H,-Statf}(AS) \wedge \Box\ (Abs(CS) = AS \wedge Abs(CS') = AS') \end{aligned} \tag{10.5}$$

Ignoring the idle state predicates in the following, we first deduce that the concurrent system $\texttt{SPAWN}_{n,H}$ satisfies the invariant $Inv(CS)$ at all times in an extra lemma. (This proof is similar to the proof of RG rule 6.1. Both the invocation and return step of each process trivially propagate the RG conditions whereas internal steps preserve them according to (10.1).) After applying this lemma in (10.5), we can add formula

$$\exists\ AS.\ \Box\ (Abs(CS) = AS \wedge Abs(CS') = AS')$$

to the antecedent of (10.5), since the abstraction function Abs is total over invariant states. More precisely, we can directly deduce that

$$\Box\ (Inv(CS) \wedge Inv(CS')) \rightarrow \exists\ AS.\ \Box\ (Abs(CS) = AS \wedge Abs(CS') = AS')$$

by using the unique sequence of *function* values $Abs(CS)$ for the existentially quantified variable AS. This gives us the abstract state variable AS that we then use for the existential quantifier in the succedent of (10.5) and property (10.4) ensures that $AInit_H(AS)$ holds now.

With some simple rewriting, it finally remains to prove the sublemma

$$\begin{aligned} &Inv(CS), \texttt{SPAWN}_{n,H}(CS), \Box\ (R{\leq_n}(CS', CS'') \wedge (Inv(CS') \rightarrow Inv(CS''))), \\ &\Box\ (Abs(CS) = AS \wedge Abs(CS') = AS') \\ \vdash\ &\texttt{ASPAWN}_{n,H,-Statf}(AS) \end{aligned}$$

by induction over n and symbolic execution (Theorem 2.2). The base case follows directly from the preconditions (10.1)/(10.2) for process 0. The inductive step exploits the compositionality rule (1.1) to rewrite the interleaved components using either the induction hypothesis, or the local RG/refinement proof obligations (10.1)/(10.2) and then uses similar arguments as the proof of RG rule 6.1 to establish that RI and the abstraction property can be always assumed locally. This concludes the proof of ii) and the soundness proof of the refinement-based proof method. □

State-Local Proof Obligation for Linearizability In Theorem 10.1 we can use the state-local RG rule 6.2 instead of rule 6.1 to establish the RG conditions for refinement. Then the following state-local version of our proof obligation (10.2) for refinement is also sufficient for linearizability:

$$\begin{aligned} &Idle(LS), \texttt{COP}(I, In; LS, \mathcal{S}, Out), \Box\ Out' = Out'', \\ &\Box\ (LS' = LS'' \wedge R(LS', \mathcal{S}', \mathcal{S}'') \wedge Inv(LS, \mathcal{S}) \wedge Inv(LS', \mathcal{S}')) \\ \vdash\ &\exists\ AS.\ \texttt{AOP}(I, In; AS, Out) \wedge \Box\ (Abs(\mathcal{S}) = AS \wedge Abs(\mathcal{S}') = AS') \end{aligned} \tag{10.6}$$

The soundness proof for this reduced version is not difficult, see [59].

10.2. Case Study: A Multiset with Fine-Grained Locking

This section illustrates the application of our refinement-based proof method for linearizability. As a non-trivial case study we use the blocking multiset implementation of [29]: It applies fine-grained locking and implements an operation `INSERTPAIR` that adds two elements to the multiset such that it appears as if both elements were inserted atomically. Such operations are relevant for a variety of concurrent data structures, such as file systems and storage systems that require multiple resources for successful completion according to [31]. We have chosen this case study for several reasons: It is a nice example that shows how ownership annotations can be used to significantly simplify specifications and to reduce the verification effort. Due to these annotations, the RG specifications are over the shared state only and do not consider any local state.

Furthermore, while the authors of [29] claim that refinement-based approaches can not easily handle the linearizability of `INSERTPAIR`, we give a simple abstraction function and a compositional refinement proof for linearizability. Moreover, the operation `LOOKUP` has a potential linearization point that depends on the behavior of other processes. Different from other approaches [26], we do not require backward simulation to deal with such potential linearization points. The linearizability proofs of the multiset case study have been mechanized in KIV; these are available online [60]. The following description corresponds to our journal article [99].

10.2.1. The Multiset with Fine-Grained Locking

Figure 10.1 shows the array-based multiset implementation. Array Ar contains elements of type *slot* where each slot consists of an arbitrary element (with selector function .elt) and a status that represents the state of insertion (with selection function .st). The status of a slot can be either *empty*, *reserved* or *full*. An element x is considered to be in the multiset if there is some array slot with element x and status *full*.

The lookup operation `LOOKUP`(x) tests whether a given element x is in the multiset by traversing the array and checking each slot for element x with status *full* (at location $L3$). Before this check is done, a slot is locked and released afterwards to prevent unexpected concurrent changes of the tested slot.

The operation `INSERTPAIR`(x, y) adds two elements to the multiset. First, it reserves an empty slot by calling operation `FINDSLOT` which traverses the array similarly to a lookup. If it cannot find an empty slot it returns the length of the array N to indicate failure. In this case, the calling insertpair operation also fails and returns *false* at I3. Otherwise, findslot is called a second time to reserve a second empty slot. If this fails, the slot reserved during the first call is released at location I7 and `INSERTPAIR` returns false at I8. If two slots i and j have been found, they are assigned the input values x and y at I10 and I11, respectively. To finish insertion, both slots are first locked, then changed to status *full* (at I14 and I15) and finally unlocked. Unlocking the elements makes the insertion visible to other processes *non-atomically*.

```
status: empty | reserved | full;
slot: (.elt: elem, .st: status);
var Ar: array[0..N-1] of slot

LOOKUP(x: elem): bool
L0 var i: int;
L1 for (i := 0; i < N; i++) {
L2   lock(Ar[i]);
L3   if (Ar[i].elt = x && Ar[i].st = full) {
L4     unlock(Ar[i]);
L5     return true;
L6   } else unlock(M[i]);
L7 };
L8 return false

FINDSLOT(x: elem): int
F0 var i: int;
F1 for (i := 0; i < N; i++) {
F2   lock(Ar[i]);
F3   if (Ar[i].st = empty) {
F4     Ar[i].st := reserved;
F5     unlock(Ar[i]);
F6     return i;
F7   } else unlock(Ar[i]);
F8 };
F9 return N
```

```
INSERTPAIR(x:elem, y:elem): bool
I0  var i,j: int;
I1  i := FindSlot(x);
I2  if (i = N) {
I3    return false
I4  };
I5  j := FindSlot(y);
I6  if (j = N) {
I7    Ar[i].st := empty;
I8    return false;
I10 Ar[i].elt := x;
I11 Ar[j].elt := y;
I12 lock(Ar[i]);
I13 lock(Ar[j]);
I14 Ar[i].stt := full;
I15 Ar[j].stt := full;
I16 unlock(Ar[i]);
I17 unlock(Ar[j]);
I18 return true
```

Figure 10.1.: The Multiset Operations `LOOKUP`, `INSERTPAIR` and `FINDSLOT`.

Extensions. In [99], we have considered an abstract specification of the FINDSLOT operation (as in the original paper [29]) that atomically chooses an arbitrary array slot with status empty. Here, we consider a concrete implementation that traverses the array starting at its lower end. Our previous atomic specification suggests that different implementations of FINDSLOT are also possible, e.g., an implementation that traverses the array starting at its upper end. We have also taken a linearizable insert operation INSERT(x) (with fine-grained locking) into account that adds just one element, but we leave it away here, since having such an operation adds no further notable challenges. Finally, our linearizability result for our wait-free multiset from Section 9.4 suggests that a (blocking) delete operation could be also used in the version with fine-grained locking, thus solving an open problem in [99].

10.2.2. Challenges of Verifying the Multiset Linearizable

The problems of verifying this case study linearizable are as follows:

1) Inserting two elements atomically adds a significant amount of complexity to the implementation according to [31]. It is counter-intuitive to think of exactly one location as its linearization point, since the insertion of the two elements is actually implemented by several non-atomic instructions. Hence, finding an abstraction function for a refinement proof can be challenging as [29] already pointed out.

2) Similar to our wait-free lookup, the blocking lookup operation has a potential linearization point which can not be statically matched with a specific instruction of its code. This is because when lookup fails to find an element x, its linearization can only happen before it passes the first slot of the array. After passing the first slot, element x might be concurrently inserted below the current search index and linearizing to false becomes impossible as x always remains in the abstract multiset. Hence, the lookup operation must potentially linearize to false before it passes the first array slot. This potential linearization, however, must be possibly revised if x is concurrently inserted ahead of the current position of lookup.

In such cases, where a linearization point can not be statically matched with a specific code instruction, a backward simulation is usually required. (Remember that our generic proof method uses step-local backward simulation, which could also be used here.) However, our refinement-based proof method can handle such cases in a more convenient manner as we explain below.

As an aside, also note that concurrent writing instructions to locked slots are possible here: A lookup process might lock a reserved slot that gets concurrently modified by the reserving insertpair operation. This can add extra complexity to a proof since it has to ensure that these writings do not violate correctness. Typically, locked resources are not changed concurrently at all.

10.2.3. The Abstract Multiset Specification

To prove linearizability, we must define an abstract semantics for our implementation. The abstract specification for the blocking multiset is similar to the one from Section

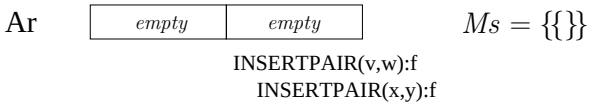
9.4.3. The abstract state Ms of type $mset$ is an algebraic multiset of elements which is initialized to be empty according to predicate $AInit(Ms)$.

The abstract lookup relation $ALookUp$ is defined just as it is defined in Section 9.4.3 for the wait-free multiset. The abstract relation $AInsertPair$ either adds both x and y to the current multiset and returns true, or it fails non-deterministically with output false, leaving the multiset unchanged:

$$AInsertPair(x, y, Ms, Ms', B) \equiv Ms' = Ms \cup \{\{x, y\}\} \wedge B \vee \neg\, B \wedge Ms' = Ms$$

Strengthening operation $AInsertPair$ to only fail when a bound on the number of elements in the abstract multiset is reached (typically the size of the array) makes the implementation non-linearizable. To better understand why, consider the following execution for an array of length 2 that is initially empty: Two distinct processes p and q reserve the two slots with their first call to findslot, respectively:

Then the second call to findslot fails for both processes which subsequently release their reserved slots and return with false.



On the abstract level, however, the first abstract insertpair operation succeeds and returns true, since there is still room for elements in Ms. Hence, the concurrent execution above would not be linearizable w.r.t. this stronger semantics.

10.2.4. The Concrete Multiset Specification in RGITL

Figure 10.2 shows the formal specification of the concrete multiset algorithms in RGITL. It is close to the pseudo code from Figure 10.1 and uses the same data structures; the minor syntactic differences result from the specific constructs of the abstract programming language of the logic. In particular, since there is no return statement in the language, the local variable *Found* is introduced and assigned to an additional output parameter *Out* at the end of each program.

Similar to the proof scripts of [29],[3] we annotate the ownership of array slots by the current process in the program. These annotations help to improve both specification and verification significantly. The auxiliary variable $O\colon \mathbb{N} \to \mathbb{N} \cup \{none\}$ encodes the owner process of a particular array slot. Value *none* indicates that an array slot currently has no owner. Ownership is always updated in parallel with an actual program instruction: A process executing `INSERTPAIR` owns exactly its reserved array slots until it sets the status of both to *full*, or it owns a single slot until it sets the slot status back to *empty* again when it fails to find a second empty slot.

[3] Available at `http://qed.codeplex.com/`.

```
LOOKUP_p(x; Ar, O, Out) {
 let Found = false, Ix = 0 in {
  while ¬ Found ∧ Ix < N do {
   LOCK_p(Ix; Ar);
   if Ar[Ix].elt = x  ∧ Ar[Ix].st = full then
    Found := true;
   UNLOCK(Ix; Ar);
   Ix := Ix + 1};
  Out := Found}}

INSERTPAIR_p(x, y; Ar, O, Out) {
 let Found = false, Ix, Jx in {
  FINDSLOT_p(Ar, O, Ix);
  if Ix < N then {
   FINDSLOT_p(Ar, O, Jx);
   if Jx < N then {
    Ar[Ix].elt := x;
    Ar[Jx].elt := y;
    LOCK_p(Ix; Ar);
    LOCK_p(Jx; Ar);
    Ar[Ix].st := full;
    Ar[Jx].st := full;
    Found := true, O(Ix) := none, O(Jx) := none;
    UNLOCK(Ix; Ar);
    UNLOCK(Jx; Ar)
   } else {
    Ar[Ix].st := empty, O(Ix) := none}};
  Out := Found}}

FINDSLOT_p(Ar, O, Ix) {
 let Found = false, Ix_0 = 0 in {
  while ¬ Found ∧ Ix_0 < N do {
   LOCK_p(Ix_0; Ar);
   if Ar[Ix_0].st = empty then {
    Ar[Ix_0].st := resrvd, O(Ix_0) := p;
    UNLOCK(Ix_0; Ar);
    Found := true
   } else {
    UNLOCK(Ix_0; Ar);
    Ix_0 := Ix_0 + 1}};
  Ix := Ix_0}}

LOCK(p, Ix; Ar) {
 await Ar[Ix].lck  = none;
  Ar[Ix].lck := p}

UNLOCK(Ix; Ar) {
 Ar[Ix].lck := none}
```

Figure 10.2.: RGITL Specification of LOOKUP and INSERTPAIR Operations with Ownership Annotations (Shaded).

As a nice effect of these annotations, the state of the concrete specification consists of the globally visible variables Ar and O only. No properties about the local state of one (or several processes) is necessary. Local variables can be introduced with **let** and are thus hidden by existential quantification from specifications. Hiding the local state from being visible in the interface of procedures is not always possible, but it is possible here, since the relevant information about the local states can be made globally visible using the auxiliary ownership variable.

10.2.5. An Abstraction Function with Ownership

In our refinement-based approach for proving linearizability, we have to find an abstraction function $Abs\colon cstate \to astate$ that maps each concrete state of the implementation to an abstract state. Moreover, we have to show that the concrete implementation refines the abstract atomic multiset operations modulo stuttering. Hence, the abstract multiset must remain unchanged during an execution of `INSERTPAIR` as long as the linearization point of the operation is not reached yet. When the linearization point is executed, the state changes immediately to a multiset containing both elements.

One could try to use a naive abstraction function like

$$cnt(x, Abs(Ar)) \;= \mid \{n \mid n < N \wedge Ar[n].elt = x \wedge Ar[n].st = full\} \mid$$

where Abs is implicitly defined by function $cnt : \text{elem} \times mset \to \text{nat}$ which counts the number of occurrences of an element in Ar with status full. As [29] already mentioned, this abstraction function is not useful. With this function the abstract multiset would change twice during the execution of `INSERTPAIR` (in I14 and I15), which violates linearizability.

Alternatively, one could try considering only those elements to be in the multiset that have status *full* and which are not locked. This idea is even worse for two reasons: First, unlocking in `INSERTPAIR` is not atomic either and just moves the problem to locations I16 and I17. Second, `LOOKUP` would change the multiset during traversal of the array by locking and unlocking slots, which would also violate linearizability.

Based on the introduced ownership annotations, we are able to give a simple definition of Abs which is close to the naive function above.

$$\begin{aligned} &cnt(x, Abs(Ar, O)) = \\ &\qquad \mid \{n \mid n < N \wedge M[n].elt = x \wedge M[n].st = full \wedge O(n) = none\} \mid \end{aligned}$$

That is, the number of occurrences of an element x in the multiset $Abs(Ar, O)$ corresponds to the number of occurrences of x in the array Ar with status *full* and no owner. Hence, when `INSERTPAIR` atomically gives up ownership of the two slots Ix and Jx, it linearizes with true. This abstraction function is an intuitive solution to problem 1) above.

In [101], we have introduced a more complex abstraction function without using ownership. The definition there takes the existence of another reserved slot into account. This caused some complications to ensure that when the status of a reserved slot is

reset to empty in IP7 (see Figure 10.1), the abstract multiset Ms remains unchanged. These problems no longer exist with the abstraction function that uses ownership, since the elements of array slots that do not have status full are simply not in the multiset.

10.2.6. Instantiating the Refinement-Based Proof Method

Applying our refinement-based proof method (Theorem 10.1) for linearizability first requires to prove premises of RG rule 6.1 for suitable instances of predicates $Init$, $Idle_p$, Inv, R_p and G_p for the multiset. Second, the refinement proof obligation (10.2) that implies linearizability must be shown. Together, Theorem 10.1 then ensures that the multiset is linearizable. We start by describing the instances of the RG predicates for the multiset in the following, their use in the compositional rely-guarantee proofs is explained afterwards, then we sketch the refinement proof.

The initial concrete overall system state is defined by predicate $Init$ such that all array slots have status empty and the ownership function is $\lambda\ m.\ none$. In idle states, a process p does not own any slot of the shared array.

$$Idle_p(Ar, O) \equiv \forall\ n.\ n < N \rightarrow O(n) \neq p$$

Predicate Inv is instantiated with two simple invariants: empty array slots are never owned, and owned full slots are locked by their owner.

$$\begin{aligned} &Inv(Ar, O) \equiv \\ &\quad \forall\ n.\ n < N \rightarrow \quad (Ar[n].st = empty \rightarrow O(n) = none) \\ &\qquad\qquad\qquad \wedge\ (Ar[n].st = full \wedge O(n) \neq none \rightarrow O(n) = Ar[n].lck) \end{aligned}$$

The rely predicate R_p is slightly more difficult.

$$\begin{aligned} &R_p(Ar', O', Ar'', O'') \equiv \forall\ n.\ n < N \rightarrow \\ &\quad (Ar'[n].lck = p \leftrightarrow Ar''[n].lck = p) \wedge (O'(n) = p \leftrightarrow O''(n) = p) \\ &\wedge\ (\quad Ar'[n].st = full \vee O'(n) = p \vee Ar[n].lck = p \wedge Ar[n].st = empty \\ &\qquad \rightarrow Ar'[n].elt = Ar''[n].elt \wedge Ar'[n].st = Ar''[n].st) \\ &\wedge\ (Ar'[n].st = full \wedge O'(n) = none \rightarrow O''(n) = none) \end{aligned}$$

The first two conjuncts state that array slots which are locked/owned by p remain locked/owned by p over an environment transition. (This is a generic property of locked/owned resources that is independent from the multiset application.) The third conjunct states that neither the element nor the status of array slots that are full or owned by process p are changed concurrently. (Recall that we do not consider a delete operation here.) The same also holds for slots that are locked by p and have status empty. The last conjunct states that each full slot without an owner does not get an owner concurrently.

Finally, the guarantee G_p is canonically defined as $\bigwedge_{q \neq p} R_q$.

10.2.7. Proving RG Assertions for the Multiset

The predicate logic premises of rule 6.1 are simple to prove with the instances above. It remains to prove the central proof obligation (10.1) which is an RG assertion for partial correctness for each multiset operation. The RG assertion for $\texttt{LOOKUP}_p$ can be easily shown using symbolic execution of sequential programs (see Section 4.3) and induction (4.3). The compositional proof of $\texttt{INSERTPAIR}_p$ is sketched in the following:

To avoid a monolithic proof, we split the proof in three sublemmas for the main method $\texttt{INSERTPAIR}_p$ and its two subcalls to $\texttt{FINDSLOT}_p$, respectively. That is, we first prove two suitable RG assertions for the calls to $\texttt{FINDSLOT}_p$ and then use the corresponding post-conditions in the proof of $\texttt{INSERTPAIR}_p$. For the first call to $\texttt{FINDSLOT}_p$, we assume that the current process owns no array slot as a pre-condition, i.e., predicate $owns0_p$ holds according to the following sublemma

$$
\begin{aligned}
& Inv(Ar, O), owns0_p(O, N), \Box\ Ix' = Ix'' \\
\vdash\ & [R_p(Ar', O', Ar'', O''), G_p(Ar, O, Ar', O'), Inv, \texttt{FINDSLOT}_p(Ar, O, Ix)] \\
& \textbf{if } Ix < N \textbf{ then } owns1_p(Ix, O, N) \wedge Ar[Ix].st = resrvd \textbf{ else } owns0_p(O, N)
\end{aligned}
$$

As a post-condition the sublemma establishes that either (the operation succeeds and) process p owns exactly the reserved array slot at index Ix according to $owns1_p$, or (the operation fails and) process p still owns no slot upon termination.

Similarly, the second call to $\texttt{FINDSLOT}_p$ is handled in a sublemma where initially process p owns exactly one reserved array slot at index Ix, i.e., $owns1_p$ holds as a pre-condition.

$$
\begin{aligned}
& Inv(Ar, O), owns1_p(Ix, O, N), Ar[Ix].st = resrvd, \Box\ (Ix' = Ix'' \wedge Jx' = Jx'') \\
\vdash\ & [R_p(Ar', O', Ar'', O''), G_p(Ar, O, Ar', O'), Inv, \texttt{FINDSLOT}_p(Ar, O, Jx)] \\
& \textbf{if } Jx < N \textbf{ then } owns2_p(Ix, Jx, O, N) \wedge Ar[Ix].st = Ar[Jx].st = resrvd \\
& \textbf{else } owns1_p(Ix, O, N) \wedge Ar[Ix].st = resrvd
\end{aligned}
$$

The post-condition of the sublemma states that p either owns exactly two reserved array slots at Ix and Jx according to $owns2_p$, or if the operation fails, then process p still owns its initial reserved slot at position Ix.

With these two sublemmas the proof simplifies to the following RG assertion

$$
\begin{aligned}
& Inv(Ar, O), owns2_p(Ix, Jx, Ar, O), Ar[Ix].st = Ar[Jx].st = resrvd, \\
& \Box\ (Ix' = Ix'' \wedge Jx' = Jx'') \\
\vdash\ & [R_p(Ar', O', Ar'', O''), G_p(Ar, O, Ar', O'), Inv(Ar, O), \mathbf{IP}]\ owns0_p(O, N)
\end{aligned}
$$

where **IP** is the rest program of $\texttt{INSERTPAIR}_p$ from line IP onwards, right after the successful reservation of the second slot. This goal can be easily shown by further symbolic execution for sequential programs.

10.2.8. The Refinement Proof for Linearizability

The second part of our refinement-based proof method for linearizability requires to show a refinement that is based on the previously established RG conditions RI. This proof corresponds to proof obligation (10.2) that identifies the linearization points of the two concrete operations as one instruction in the code. It uses symbolic execution for sequential programs (according to Section 2.1.5) and Theorem 2.2, respectively.

INSERTPAIR. We exploit the compositionality of RGITL in the proof of the method `INSERTPAIR` and first prove two sublemmas for $\texttt{FINDSLOT}_p$, see below. With these two sublemmas, the calls to findslot are replaced by applying rule (1.1), and the refinement proof for $\texttt{INSERTPAIR}_p$ becomes straightforward: It linearizes to true with the instruction that atomically resets the ownership of Ix and Jx to none. All other steps refine mere abstract stutter steps of $\texttt{AOP}_p$.

The first sublemma ensures that starting with no owned slots, $\texttt{FINDSLOT}_p$ never changes the abstract representation and terminates with owning exactly one slot at index Ix.

$$
\begin{aligned}
&\mathit{owns0}_p, \texttt{FINDSLOT}_p(Ar, O, Ix), \Box\ RI, \Box\ Ix' = Ix'' \\
&\Box\ (\mathit{Abs}(Ar, O) = Ms \wedge \mathit{Abs}(Ar', O') = Ms') \\
&\vdash \Box\ (\textbf{if last then}\ (Ix < N \rightarrow \mathit{owns1}_p(Ix, O, N) \wedge Ar[Ix].st = \mathit{resrvd})\ \textbf{else}\ Ms = Ms')
\end{aligned}
$$

If process p fails to find an empty slot (i.e., $Ix = N$ in the last state), then a trivial post-condition suffices to conclude that the current execution of $\texttt{INSERTPAIR}_p$ corresponds to a failing abstract run modulo stuttering: The linearization to false then occurs with the last assignment that sets the output to false.

The second sublemma requires a slightly stronger post-condition.

$$
\begin{aligned}
&\mathit{owns1}_p(Ix, O, N), \texttt{FINDSLOT}_p(Ar, O, Jx), \Box\ RI, \Box\ (Ix' = Ix'' \wedge Jx' = Jx''), \\
&\Box\ (\mathit{Abs}(Ar, O) = Ms \wedge \mathit{Abs}(Ar', O') = Ms') \\
&\vdash \Box\ (\textbf{if last then if}\ Ix < N \\
&\qquad\qquad \textbf{then}\ \mathit{owns2}_p(Ix, Jx, O, N) \wedge Ar[Ix].st = Ar[Jx].st = \mathit{resrvd} \\
&\qquad\qquad \textbf{else}\ \mathit{owns1}_p(Ix, O, N) \wedge Ar[Ix].st = \mathit{resrvd} \\
&\qquad \textbf{else}\ Ms = Ms')
\end{aligned}
$$

If the second call to `FINDSLOT` fails, we need to know that the current process p still owns one reserved slot upon termination. This ensures that the following execution of `INSERTPAIR`, after the second call to findslot fails, corresponds to abstract stutter steps and the linearization happens again with the last assignment that sets the output to false.

LOOKUP. The proof for the lookup method is more challenging due to its *potential* linearization point. Recall that when $\texttt{LOOKUP}_p(x)$ returns false, its linearization can only happen before it passes the first slot of the array (see problem 2) above). After passing the first slot, element x might be concurrently inserted below the current position of lookup and linearizing with false becomes impossible. Hence, a lookup method must be

potentially linearized with false before passing the first array slot, but this linearization must be revised if x is concurrently inserted in a part of the array that LOOKUP has not scanned yet.

Therefore, the refinement proof for $\texttt{LOOKUP}_p(x)$ has two cases:

i) If x is in the array before the loop is entered (loop index $Ix = 0$), then

$$\exists\, n < N.\ Ar[n].elt = x \wedge Ar[n].st = \mathit{full} \wedge O(n) = \mathit{none}$$

holds and the scan eventually finds x, since there is no delete operation, and the property above is preserved throughout symbolic execution (R_p states that both element and status of full slots are not concurrently changed and that full unowned slots remain unowned).

ii) Element x is not in the array initially and the proof executes a potential linearization of the abstract program to *false* simultaneously with the transition that evaluates the loop condition. This is possible due to the non-determinism of the star operator, which permits execution of either a linearization step that leaves Ms unchanged (zero iterations of the first $\mathbf{bskip}^*$ in $\texttt{AOP}_p$) or a stutter step. After execution of the first four steps of $\texttt{LOOKUP}_p$ (the remainder of the program is abbreviated as $\mathbf{L}$ which performs the test for the locked slot next) we get the following goal

$$\begin{array}{l} \quad \mathit{Inv}, \mathit{Idle}_p, \mathit{Found} = \mathit{false}, Ix < N, \mathbf{L},\ \Box\ (\mathit{RI} \wedge \mathit{Found}' = \mathit{Found}'' \wedge Ix' = Ix''), \\ \quad \Box\ (\mathit{Abs}(Ar, O) = Ms \wedge \mathit{Abs}(Ar', O') = Ms') \\ \vdash\ \mathbf{bskip}^*;\ \mathit{Out} := \mathit{Found}, \\ \quad \mathbf{bskip}^*;\ \{\mathit{ALookUp}(x, Ms, Ms', \mathit{Found}') \wedge\ \ldots\};\ \mathbf{bskip}^*;\ \mathit{Out} := \mathit{Found} \end{array}$$

The two formulas in the succedent correspond to the potential linearization: The first formula indicates that *ALookUp* has already linearized to false (the local output *Found* is *false* now). The second formula indicates that the linearization point was not reached yet, and only stutter steps have been executed so far.

With the instruction that successfully finds x, the potential linearization is revised by weakening the first program from the succedent and the algorithm linearizes to true. (We know from *Inv* and predicate Idle_p that the slot has no owner when this test succeeds, i.e., the element is in the abstract multiset.) If the test fails, we have two cases: either the loop must be reiterated (if $Ix < N$ still holds after incrementation) and the proof goal is closed with an inductive argument (basically we use induction over sequential programs here, after instantiating the local output value of the abstract program with *Found*), or the end of the array is reached ($Ix = N$) without finding x. In the latter case, the initial linearization to false was correct.

This concludes our proof sketch for linearizability of the multiset with fine-grained locking. Full proof details are available at [59].

10.3. Summary

In this chapter we have introduced a compositional proof method for the important class of linearizable algorithms with internal linearization points, including a state-

local version of our proof obligations that avoids reasoning over the full program state. The correctness proof of our method(s) w.r.t. the original definition of linearizability/possibilities is mechanized in KIV. Different from other refinement-based approaches, our method does not require backward simulation which helps to simplify the proofs. For an extended discussion on related work see Chapter 13.

Furthermore, we have applied our proof method to verify linearizability of a challenging multiset implementation with fine-grained locking according to [29]. We show how ownership annotations can be used in this case to avoid reasoning over local states and to handle the linearization of several non-atomic insert instructions in a convenient way. The correctness proofs for the multiset are mechanized in KIV.

We have applied our state-local refinement-based proof method to various other algorithms with internal linearization points such as the lock-free stack [102] – with modification counters [42] and with hazard pointers [68] including non-atomic reads/writes of hazard pointers (see Section 12.2) – the MS queue [69] (under garbage collection and with hazard pointers, see Section 12.2), the more efficient lock-free queue [26] or the non-blocking set [70] (see Section 11.4). We have also verified the essential liveness property of lock-freedom for all of these algorithms using our compositional proof method for lock-freedom which we introduce in the next chapter.

11. A Proof Method for Lock-Freedom

The previous two chapters have dealt with the verification of the central *safety* property of linearizability. This chapter focuses on the verification of *liveness* (progress) aspects of non-blocking data structure implementations that do not use locks. In Section 11.1, we define the three central non-blocking progress properties – wait-freedom, lock-freedom, obstruction-freedom – and illustrate them with simple examples. Subsequently, we introduce a compositional proof method for the important global progress property of lock-freedom [67] in Section 11.2, including a state-local version of our proof obligations.

To compositionally prove lock-freedom we combine RG reasoning with temporal logic as follows: RG reasoning first establishes individual rely conditions that are then used in a temporal logic termination proof for an individual process. For the termination proofs, we use an additional binary predicate U that characterizes interference conditions that are specific for termination. Section 11.3 details the soundness proof of our compositional proof method that has been mechanized in KIV [59]. In Section 11.4 we then illustrate the application of our proof method to verify a highly concurrent set data structure. This proof is also mechanized in KIV [59]. The description largely follows our journal article [89] and the basic ideas of our proof method can also be found in our article [94]. The description of the set case study is based on [81].

11.1. Non-Blocking Progress

Non-blocking progress is relevant in various application domains where the use of locks would lead to deadlock. For instance, high-availability systems are expected to run for a long time and when individual components fail, their failure should not cause a deadlock of the entire system. Another example are async-signal safe operations where handling interrupts should not lead to deadlock. Moreover, non-blocking progress can be of benefit in real-time systems where prioritized threads must progress after a finite number of own steps.

To specify non-blocking progress conditions, we first determine our underlying concurrent system model. Basically it corresponds to the system model $\texttt{SPAWN}_n$ from Section 6.1 with the following minor adaptations:

$$
\begin{array}{l}
\mathtt{SPAWN}_{n,Actf}(S)\ \{ \\
\quad \textbf{if*}\ n = 0\ \textbf{then} \\
\qquad \mathtt{PROC}_{0,Actf_0}(S) \\
\quad \textbf{else} \\
\qquad \mathtt{PROC}_{n,Actf_n}(S) \\
\qquad \|_{\mathsf{nf}}\ \mathtt{SPAWN}_{n-1,Actf}(S) \\
\}
\end{array}
$$

The state S is extended with an auxiliary function $Actf\colon nat \to bool$ where $Actf_p$ means that process $p \leq n$ is running and $\neg\ Actf_p$ captures the termination of process p.[1] The generic procedure $\mathtt{PROC}_p$ from $\mathtt{SPAWN}_n$ is now implemented such that process p infinitely often executes the following steps:

$$
\begin{array}{l}
\mathtt{PROC}_{p,Act}(S)\ \{ \\
\quad \textbf{while}\ \mathit{true}\ \textbf{do}\ \{ \\
\qquad Act := \mathit{true}; \\
\qquad \mathtt{COP}_p(S); \\
\qquad Act := \mathit{false} \\
\quad \}\ \}
\end{array}
$$

First, it sets its activity flag Act to true, then it executes internal steps $\mathtt{COP}_p$ after which it resets its activity flag to false again.

Moreover, since we only consider non-blocking algorithms here, we can assume that $\mathtt{COP}_p$ transitions never block

$$\mathtt{COP}_p(S) \vdash \Box\ \neg\ \textbf{blocked} \tag{11.1}$$

This condition enforces that instances of $\mathtt{COP}_p$ must not use the **await** statement. As a simple consequence, the concurrent system $\mathtt{SPAWN}_{n,Actf}$ never runs into a deadlock, but livelocks are still possible.

Now to the definitions of the three non-blocking progress conditions. The first two progress conditions, wait-freedom and obstruction-freedom, are defined in terms of an individual operation $\mathtt{COP}_p$.

Wait-free operations terminate in a finite number of own steps.

Definition 11.1 (Wait-Freedom)
Each individual operation $\mathtt{COP}_p$ *eventually terminates (in a safe environment that never violates rely conditions* R_p*):*

$$\mathtt{COP}_p(S), \Box\ R_p(S', S'') \vdash \Diamond\ \textbf{last}$$

[1] Alternatively, we could have used a history enhanced concurrent system $\mathtt{SPAWN}_{n,H}$ where (global) progress can be expressed in terms of the number of pending invocations in H. For simplicity, however, we prefer to express termination in terms of a boolean flag here.

Note that by explicitly taking the individual rely conditions R_p into account, our Definition 11.1 also provides a proof method for wait-freedom based on RG reasoning. The individual rely conditions R_p that are necessary to prove termination of $\texttt{COP}_p$ must be established first in the execution context of $\texttt{SPAWN}_{n,Actf}$, e.g., by applying RG rule 6.1. To better focus on the core aspects, we only use rely conditions in the following definitions (to represent the execution context), but leave out pre-/post-conditions and invariants. These can be easily added if required in concrete liveness proofs.

Example 11.1 (Wait-Freedom)
Our multiset implementation from Figure 9.2 is wait-free, since all of its methods terminate in a safe environment. To prove this, we require the fact that the length of the array is never concurrently changed by any multiset method as a rely condition in each individual proof of termination.

The push and pop operations of the concurrent stack implementation from Figure 8.4 are not wait-free, since the repeated concurrent change of the top-of-stack pointer can lead to individual starvation even in a safe environment.

Similarly we define obstruction-freedom: Obstruction-free operations terminate in a finite number of own steps once they execute in isolation.

Definition 11.2 (Obstruction-Freedom)
Each operation $\texttt{COP}_p$ terminates when it eventually runs without interference from its (safe) environment:

$$\texttt{COP}_p(S), \Box\ R_p(S', S'') \vdash (\Diamond\ \Box\ S' = S'') \rightarrow \Diamond\ \textbf{last}$$

Example 11.2 (Obstruction-Freedom)
The (operations of the) CAS-based counter in Figure 8.3 and the CAS-based stack from Figure 8.4 are obstruction-free. This is simple to prove: Once their environment satisfies $\Box\ N' = N''$ and $\Box\ Top' = Top''$, respectively, each operation terminates after at most one retry of its CAS-loop.

In contrast, lock-based algorithms (e.g., the multiset with fine-grained locking from Figure 10.1) are not obstruction-free, since executing an operation that waits for a lock in an empty environment does not lead to its termination.

Different from wait-freedom and obstruction-freedom, which we define in terms of individual operations, the definition of lock-freedom takes all interleaved operations into account.

Definition 11.3 (Lock-Freedom)
The concurrent system $\texttt{SPAWN}_{n,Actf}$ is lock-free if infinitely often one of its interleaved operations progresses:

$$\texttt{SPAWN}_{n,Actf}(S),\ \Box\ (S' = S'' \wedge Actf' = Actf'') \vdash \Box\ \Diamond\ P{\leq_n}(Actf, Actf')$$

where $P{\leq_n}(Actf, Actf') \equiv \exists\ p \leq n.\ P_p(Actf, Actf')$
and $P_p(Act, Actf') \equiv Actf_p \wedge \neg\ Actf'_p$.

Definition 11.3 specifies a system-wide progress transition $P\leq_n$ simply by requiring that one of the spawned processes p resets its activity flag from true to false which constitutes an individual progress step P_p of process p. Note that in a concurrent system that repeatedly invokes data structure operations such as $\texttt{SPAWN}_{n,Actf}$, lock-freedom does not exclude the starvation of individual processes, but it ensures system-wide progress, even under unfair scheduling where some processes might never be scheduled again. Hence, a lock-free concurrent system where only a finite number of method invocations occur, eventually terminates.

Example 11.3 (Lock-Freedom)
The CAS-based counter from Figure 8.3 is lock-free. Informally, this is because a process p is only forced to retry its CAS-loop when some other process q executes a step that leads to the termination of q (here, by executing a successful CAS). However, a formal argument for the absence of global livelock is not easy to get and a compositional proof method for lock-freedom is desirable.[2]

As a final simple example, consider the following interleaved system [28]

$$\{\,\textbf{while}\ \mathit{true}\ \textbf{do}\,\{\,\textbf{if}\ B\ \textbf{then break}; \textbf{else}\ B := \neg\ B\}\}$$
$$\|_{\mathsf{nf}}\ \{\,\textbf{while}\ \mathit{true}\ \textbf{do}\,\{\,\textbf{if}\ \neg\ B\ \textbf{then break}; \textbf{else}\ B := \neg\ B\}\}$$

where two processes repeatedly toggle a shared boolean variable B. This system is not lock-free since the two processes can mutually cause a retry of their loops that leads to global livelock. However, we can easily prove that each individual operation satisfies the obstruction-free condition as it terminates when its local environment eventually leaves B unchanged.

11.2. Proof Method: Termination Under Interference

In the following, we give a generic and compositional proof method for lock-freedom: It is generic as it applies to a wide range of lock-free algorithms and compositional, since it reduces the proof of lock-freedom for $\texttt{SPAWN}_{n,Actf}$ to two simple termination proof obligations for its individual operations $\texttt{COP}_p$. These are explained in the following.

The first proof obligation ensures that $\texttt{COP}_p$ terminates whenever it does not suffer from critical interference from its environment. This interference is specified with an additional binary predicate U ("unchanged"):

$$\texttt{COP}_p(S), \Box\ R_p(S', S'') \vdash \Diamond\ \Box\ U(S', S'') \rightarrow \Diamond\ \textbf{last} \tag{11.2}$$

The binary relation $U\colon \mathit{state} \times \mathit{state}$ is required to be reflexive and transitive. It captures under which conditions termination of $\texttt{COP}_p$ can be guaranteed. These conditions are specific for termination and cannot be covered by rely conditions. While rely conditions R_p are safety properties that are never violated ($\Box\ R_p(S', S'')$), the progress conditions U can be repeatedly violated during the execution of an operation.

[2] [73] dedicates a full paper to derive the slightly stronger property of $\Box\ \Diamond\ N' = N + 1$ for a simple CAS-based counter in a compositional way.

Example 11.4 (Environment Conditions for Termination)
For the CAS-based counter from Figure 8.3, critical interference is the concurrent increment of N, i.e., U would be instantiated as the identity relation over N and termination can be easily shown for an individual increment operation whenever the environment leaves N unchanged.

Similarly, for the CAS-based stack from Figure 8.4, critical interference is the concurrent change of the top-of-stack pointer and thus U corresponds to the identity relation over the top-of-stack pointer Top.

In both examples, the proof of (11.2) is straight-forward using symbolic execution for sequential programs (Section 2.1.5). To deal with (CAS-)loops that are reiterated until no interference is encountered, we extract a counter for well-founded induction from the liveness property $\Diamond\ \Box\ U(S', S'')$ *using the generic induction rule (2.4).*

The second proof obligation enforces that each $\texttt{COP}_p$ causes only a bounded amount of interference, i.e., it violates U only a finite number of times in its *own steps*. Formally, executions where U is violated by transitions of $\texttt{COP}_p$ infinitely often can be disallowed as follows:

$$\texttt{COP}_p(S), \Box\ R_p(S', S'') \vdash \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last} \tag{11.3}$$

Alternatively, this can be also understood by contraposition $\Box \neg\ \mathbf{last}\ \rightarrow \Diamond \Box\ U(S, S')$, i.e., in infinite executions a program eventually causes no interference. It is important to note that we use U here to restrict *program transitions* of $\texttt{COP}_p$, in contrast to (11.2) where it restricts environment transitions.

Example 11.5 (Program Conditions for Termination)
For the CAS-based counter from Figure 8.3, the increment operation changes N at most once with its successful CAS instruction and this step leads to its termination. Similarly, a push/pop operation in Figure 8.4 changes the top-of-stack pointer at most once (with a CAS) and afterwards it terminates. Thus the identity relations over N/Top are the right instances that allow us to prove termination in both proof obligations (11.2) *and* (11.3).

In both examples, the proof of the second proof obligation (11.3) uses induction over the liveness property $\Diamond \neg\ U(S, S')$ *which gives the number of steps until a transition from S to* S' *is executed that violates* $U(S, S')$*; this transition then immediately leads to termination as required. In examples where termination can not be guaranteed after a first violation of U has happened (see Section 12.1), we have to induct again over the number of steps until a further violation occurs.*

As already explained, predicate U plays two different roles in each of the two proof obligations for termination: On the one hand, in (11.2) it restricts interference from the *environment* of an individual operation. On the other hand, it restricts the behavior of an individual *program* in (11.3). Both proof obligations (11.2) and (11.3) together with an RG assertion for partial correctness (according to rule 6.1) to establish RG conditions for safety, are sufficient for lock-freedom:

Theorem 11.1 (Soundness of Proof Method for Lock-Freedom)
Given the premises of RG rule 6.1 and the local termination conditions (11.2) and (11.3) for the instances described above, it follows that the concurrent system $\texttt{SPAWN}_{n,Actf}$ *is lock-free.*

The non-trivial proof of Theorem 11.1 is detailed in the next Section 11.3. We conclude this section with discussing further aspects and versions of Theorem 11.1.

It is essential in our definition of lock-freedom to consider *unfair* interleaving, where individual processes might be preempted and never resumed again. Formally, this corresponds to the case of fairness rule (2.7) where component 1 gets discarded. (Remember that all intervals of the concurrent system model $\texttt{SPAWN}_{n,Actf}$ are infinite and non-blocking.) Never resuming a preempted process again for execution is a way to simulate its crash. However, our formalism does not consider explicit crashes.[3] As a nice side effect, our process-local termination proofs do not have to deal with possible crashes which keeps them straightforward.

Finally, our previous version of proof obligation (11.3) in [94] is more specific than our definition here. It requires that each $\texttt{COP}_p$ violates U at most *once* before termination.

$$\texttt{COP}_p(S), \Box\ R_p(S', S'') \vdash \Box\ (\neg\ U(S, S') \rightarrow \Diamond\ \textbf{last})$$

Thus it can not be applied in some cases where further violations of U are possible before an operation reaches its final state. (In Section 12.1 we give an example where termination is reached after 2 violations of U.)

State-Local Proof Obligations for Lock-Freedom. Using the state-local RG rule 6.2 in Theorem 11.1 instead of rule 6.1 leads to the following state-local versions of proof obligations (11.2)/(11.3)

$$\texttt{COP}(LS, \mathcal{S}), \Box\ (LS' = LS'' \wedge R(LS', \mathcal{S}', \mathcal{S}'')) \vdash \Diamond\ \Box\ U(\mathcal{S}', \mathcal{S}'') \rightarrow \Diamond\ \textbf{last} \qquad (11.4)$$

and

$$\texttt{COP}(LS, \mathcal{S}), \Box\ (LS' = LS'' \wedge R(LS', \mathcal{S}', \mathcal{S}'')) \vdash \Box\ \Diamond\ \neg\ U(\mathcal{S}, \mathcal{S}') \rightarrow \Diamond\ \textbf{last} \qquad (11.5)$$

for just one local state LS and the global state $\mathcal{S}$. This reduced version is also sufficient for lock-freedom. (Again, we omit pre-/post-conditions and invariants here. The versions with all additional predicates are available at [59]).

11.3. The Soundness Proof

This section describes the technical details of the proof of Theorem 11.1: The proof applies RG rule 6.1, the compositionality of the chop, star and interleaving operators (rule (1.1)), the fairness rules (2.7), (2.9) including their symmetric versions, and

[3] In particular, we do not consider a crash of the entire system (which is always scheduled again for execution since top-level environment transitions always terminate), but proving progress for a crashed system would be impossible anyway.

symbolic execution of the interleaving operator (see Appendix A.1 and Section 1.4, but only the case of unblocked steps is relevant here since the interleaved procedures neither block nor terminate).

The core correctness argument is as follows (assuming an empty overall environment and ignoring RG conditions for a moment): Consider the unfair interleaving of two components that each satisfy both properties in the succedents of (11.2) and (11.3). If w.l.o.g. component 1 infinitely often interferes through $\Box \diamond \neg\ U(S, S')$ with the second component, then its progress condition (11.3) would ensure that it will always eventually terminate. The contrary case, in which component 1 eventually never interferes, i.e., $\diamond \Box\ U(S, S')$, implies that component 2 will eventually never suffer from interference from its environment and thus it will repeatedly terminate according to proof obligation (11.2). If one component is preempted and never resumed again for execution, then the remaining component progresses, and the local progress of either of the two components also ensures global progress.

To start with the formal proof, we give some simple preliminary definitions and lemmas, mainly to enable an inductive liveness proof for the interleaving of one process $\texttt{PROC}_{n+1}$ with the other processes $\texttt{SPAWN}_{n,Actf}$. Furthermore, since each process only guarantees liveness under certain rely conditions R_p, liveness and RG behavior of the processes must be merged.

We start by strengthening the safety assumptions R_p and the system rely $R{\leq_n}$ from RG rule 6.1 to include that activity entries $Actf_p$ are local:

$$R_p^{act}(S', Actf', S'', Actf'') \equiv R_p(S', S'') \wedge Actf'_p = Actf''_p$$
$$R{\leq_n^{act}}(S', Actf', S'', Actf'') \equiv \forall\ p \leq n.\ R_p^{act}(S', Actf', S'', Actf'')$$

It is straightforward to derive that each process p never changes any activity flag other than $Actf_p$, by applying rule (1.12) on $\texttt{PROC}_m$. This property is denoted as $\lceil Actf_{=n} \rceil$ in the following. It is also simple to derive that $\texttt{SPAWN}_{n,Actf}$ never changes activity flags $Actf_k$, where k is greater than n, which we denote as $\lceil Actf_{\leq n} \rceil$.

Next we lift the two termination conditions (11.2) and (11.3) for $\texttt{COP}_p$ to one progress condition for $\texttt{PROC}_p$ in the next lemma.

Lemma 4 (Progress $\texttt{PROC}_m$)
If the termination conditions (11.2) and (11.3) hold, then each $\texttt{PROC}_p$ satisfies the following progress condition:

$$\texttt{PROC}_p(S, Actf_p) \vdash \Box\ R_p^{act} \rightarrow \Box\ (\Box\ U(S', S'') \vee \Box \diamond \neg\ U(S, S') \rightarrow \diamond\ P_p)$$

Proof The proof shifts $\Box\ R_p^{act}$ to the antecedent and then applies induction on the top-level always formula from the succedent according to rule (2.5). This allows to close repeated goals that evolve during symbolic execution by applying (lazy) induction. From (11.2) and (11.3) it follows that $\texttt{COP}_p$ satisfies the following termination property and (due to an implicit frame assumption) never changes the activity function:

$$\texttt{COP}_p \vdash \Box\ R_p \rightarrow \Box\ ((\Box\ U(S', S'') \vee \Box \diamond \neg\ U(S, S') \rightarrow \diamond\ \textbf{last}\ ) \wedge Actf = Actf') \tag{11.6}$$

We use property (11.6) to replace the procedure call $\texttt{COP}_p$ in $\texttt{PROC}_p$.[4]

Now, the central part of the proof is by symbolic execution of the loop body of $\texttt{PROC}_p$. After executing the first assignment, the activity flag $Actf_p$ is true (note that R_p^{act} ensures that it is not concurrently changed). The remaining proof then essentially simplifies to showing that the rest program

$$(\Diamond\ \textbf{last}\ \wedge\ Actf = Actf'); Actf_p := \mathit{true}$$

terminates with a progress transition P_p, which easily follows with further symbolic execution and applying induction in case that formula $\Diamond$ **last** does not terminate yet. □

The next lemma merges the progress property of $\texttt{PROC}_p$ from Lemma 4 with a corresponding RG property for $\texttt{PROC}_p$, plus the safety property $\lceil Actf_{=n} \rceil$ and the nonblocking behavior $\Box\ \neg$ **blocked**:

Lemma 5 ($\texttt{PROC}_p$ Progress and RG)
If the preconditions of RG rule 6.1 and the termination conditions (11.2) and (11.3) hold, then each $\texttt{PROC}_p$ satisfies the following properties:

$$\begin{aligned}\texttt{PROC}_p \vdash \quad & (\Box\ R_p^{act} \rightarrow \Box\ (\Box\ U(S', S'') \vee \Box \Diamond \neg\ U(S, S') \rightarrow \Diamond\ P_p)) \\ & \wedge\ (R_p \xrightarrow{+} G_p) \wedge \lceil Actf_{=n} \rceil \wedge \Box\ \neg\ \textbf{blocked}\end{aligned}$$

Proof With the preconditions of RG rule 6.1[5] it follows that

$$\texttt{PROC}_p \vdash R_p \xrightarrow{+} G_p$$

and Lemma 4 (plus formula (11.1)) then conclude the proof. □

We leave formula $\Box\ \neg$ **blocked** implicit in the following lemmas for brevity. With these prerequisites, the decomposition theorem for lock-freedom can be proved by induction over the number of interleaved processes. The main idea is to apply the compositionality rule (1.1) using suitable abstractions for each component in the inductive step. For this purpose, the specification of the concurrent system $\texttt{SPAWN}_{n,Actf}$ must be strengthened just as we have strengthened $\texttt{PROC}_p$ in Lemma 5. This results in the following sublemma:

$$\begin{aligned}\texttt{SPAWN}_{n,Actf} \vdash \quad & (\Box\ R{\leq_n^{act}} \rightarrow \Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ P{\leq_n})) \\ & \wedge\ (R{\leq_n} \xrightarrow{+} G{\leq_n}) \wedge \lceil Actf_{\leq n} \rceil\end{aligned} \tag{11.7}$$

It is simple to see that the correctness of (11.7) concludes the proof of Theorem 11.1.

The proof of Sublemma (11.7) is by structural induction over n. The base case for $n = 0$ follows from Lemma 5. In the inductive case $n + 1$, the concurrent system

[4] Its precondition $\Box\ R_p$ follows directly from the stronger premise $\Box\ R_p^{act}$. Observe that we must also use the compositionality of the chop and star operators here (for the loop).

[5] The RG assertion of rule 6.1 for $\texttt{PROC}_p$ is actually derived for the implementation of $\texttt{PROC}_p$ above, from a corresponding RG assertion for $\texttt{COP}_p$.

$\texttt{SPAWN}_{n+1,Actf}$ is $\texttt{PROC}_{n+1} \parallel_{\mathsf{nf}} \texttt{SPAWN}_{n,Actf}$ and the induction hypothesis permits to assume (11.7) for $\texttt{SPAWN}_{n,Actf}$ on arbitrary paths. Application of the compositionality rule (1.1) together with Lemma 5 as abstraction for the first component $\texttt{PROC}_{n+1}$, and the induction hypothesis for the second component $\texttt{SPAWN}_{n,Actf}$ leads to the following remaining proof obligation:

$$\begin{aligned}
&(\quad (\Box\ R^{act}_{n+1} \rightarrow \Box\ (\Box\ U(S',S'') \vee \Box \Diamond \neg\ U(S,S') \rightarrow \Diamond\ P_{n+1}))\\
&\quad \wedge (R_{n+1} \xrightarrow{+} G_{n+1}) \wedge \lceil Actf_{=n+1} \rceil)\\
\parallel_{\mathsf{nf}} &(\quad (\Box\ R^{act}_{\leq n} \rightarrow \Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ P_{\leq n}))\\
&\quad \wedge (R_{\leq n} \xrightarrow{+} G_{\leq n}) \wedge \lceil Actf_{\leq p} \rceil)\\
\vdash \Box\ &(\Box\ U(S',S'') \rightarrow \Diamond\ P_{\leq n+1})
\end{aligned}$$

A local case distinction for both interleaved components depending on whether $\Box\ R^{act}_{n+1}$ / $\Box\ R^{act}_{\leq n}$ holds, reduces the proof to four remaining lemmas (note that $\vee$ distributes over interleaving) that are proved in the following: Lemma 6 covers the main case where $\Box\ R^{act}_{n+1}$ and $\Box\ R^{act}_{\leq n}$ holds in component 1 and 2, respectively. Lemma 7 deals with the case that $\Box\ R^{act}_{n+1}$ holds in component 1 but $R^{act}_{\leq n}$ is eventually violated in component 2; the lemma for the symmetric case is similar and omitted here. Finally, Lemma 8 handles the case where each rely condition of both components is eventually violated.

Lemma 6 (Progress in Components 1 and 2)

$$\begin{aligned}
&(\Box\ (\Box\ U(S',S'') \vee \Box \Diamond \neg\ U(S,S') \rightarrow \Diamond\ P_{n+1})) \parallel_{\mathsf{nf}} (\Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ P_{\leq n}))\\
\vdash\ &\Box\ (\Box\ U(S',S'') \rightarrow \Diamond\ P_{\leq n+1})
\end{aligned}$$

This lemma contains the core argument why the two interleaved components $\texttt{PROC}_{n+1}$ (component 1) and $\texttt{SPAWN}_{n,Actf}$ (component 2) ensure global progress. In the lemma, both components are already abstracted with their corresponding progress condition from Lemma 4 and (11.7), respectively. The basic correctness argument is as follows: If component 1 would always eventually interfere through $\Box \Diamond \neg\ U(S,S')$ with the second component, then its progress condition would imply that it will always eventually progress P_{n+1}. The contrary case, in which component 1 eventually never interferes, i.e., $\Diamond \Box\ U(S,S')$, implies that one of the other processes (in component 2) can eventually always progress $P_{\leq n}$. If one component is preempted and never resumed again for execution, then the remaining component progresses, and the progress of either of the two components also ensures global progress $P_{\leq n+1}$. The formal arguments are detailed in the following proof of Lemma 6.

Proof Application of induction rule (2.5) on the always formula in the succedent gives a suitable counter for induction and a proof by contradiction: It assumes that property $\Box\ U(S',S'') \rightarrow \Diamond\ P_{\leq n+1}$ is globally violated after a finite number of steps. If a symbolic execution step does not reach this global violation point, induction can be applied to discard the goal. Otherwise, proving that the system does eventually

progress $\diamond\ P_{\leq n+1}$ in an environment that never interferes (i.e., $\square\ U(S', S'')$) gives the required contradiction.

As already explained informally, there are two possible cases for the local behavior of component 1. The first case in which component 1 always eventually interferes by means of $\square \diamond \neg\ U(S, S')$, follows from the next sublemma:

$$\diamond\ P_{n+1} \parallel_{\mathsf{nf}} (\square\ (\square\ U(S', S'') \rightarrow \diamond\ P_{\leq n})),\ \square\ U(S', S'') \vdash \diamond\ P_{\leq n+1} \tag{11.8}$$

The opposite case, where component 1 eventually no longer interferes, i.e., $\diamond \square\ U(S, S')$, follows from the following sublemma:

$$\begin{aligned} &(\diamond \square\ U(S, S') \wedge \square\ (\square\ U(S', S'') \rightarrow \diamond\ P_{n+1})) \\ &\parallel_{\mathsf{nf}} (\square\ (\square\ U(S', S'') \rightarrow \diamond\ P_{\leq n})),\ \square\ U(S', S'') \\ \vdash\ &\diamond\ P_{\leq n+1} \end{aligned} \tag{11.9}$$

Intuitively, Sublemma (11.8) is correct since either component 1 is scheduled sufficiently often and reaches the point where it progresses, or if it is preempted and never scheduled again before it reaches this point, then the remaining component 2 runs in an environment that never interferes (i.e., $\square\ U(S', S'')$) and thus eventually progresses. Formally, fairness rules (2.9) and (2.7) are used to get suitable induction hypotheses. We apply rule (2.7) to discern whether component 1 is preempted and not resumed again. The positive case is simple since the liveness property $\diamond\ P_{\leq n}$ of the remaining component 2 implies $\diamond\ P_{\leq n+1}$. In the negative case, a symbolic execution step of the interleaved formulas discerns which component is scheduled. If component 1 is scheduled and progresses now, then this step discards the goal since progress of component 1 also ensures global progress. All other cases are discarded by applying induction.

The proof of Sublemma (11.9) is similar. The main difference is that once component 1 reaches the state from which its steps no longer interfere through $\square\ U(S, S')$, one must establish that the local environment of component 2 never violates U. The positive case, in which $\square\ U(S', S'')$ holds for component 2, follows from the following sublemma:

$$(\square\ (\square\ U(S', S'') \rightarrow \diamond\ P_{n+1})) \parallel_{\mathsf{nf}} \diamond\ P_{\leq n},\ \square\ U(S', S'') \vdash \diamond\ P_{\leq n+1}$$

Its proof is symmetric to the proof of (11.8) and therefore omitted here. The case in which the environment of component 2 eventually interferes through $\diamond \neg\ U(S', S'')$ is discarded by the last sublemma:

$$\begin{aligned} &(\square\ U(S, S') \wedge \square\ (\square\ U(S', S'') \rightarrow \diamond\ P_{n+1})) \parallel_{\mathsf{nf}} (\diamond \neg\ U(S', S'')),\ \square\ U(S', S'') \\ \vdash\ &\diamond\ P_{\leq n+1} \end{aligned}$$

Its proof uses similar arguments to those of the soundness proof of RG rule 6.1 and is thus not given in detail. The main difference is that induction on a safety formula is not applicable. Instead, the symmetric versions of fairness rules (2.7) and (2.9) are used for induction. □

The next lemma considers the case in which the rely conditions $\Box\ R^{act}_{n+1}$ of component 1 hold; thus it ensures progress and RG locally, according to Lemma 5, while the rely conditions of component 2 are eventually violated locally as $\Diamond\ \neg\ R^{act}_{\leq n}$.

Lemma 7 (Progress and RG in Component 1)
With the preconditions of RG rule 6.1, the following holds:

$$\begin{aligned}
&(\quad \Box\ (\Box\ U(S', S'') \vee \Box\ \Diamond\ \neg\ U(S, S') \rightarrow \Diamond\ P_{n+1}) \\
&\quad \wedge\ (R_{n+1} \xrightarrow{+} G_{n+1}) \wedge \lceil Actf_{=n+1} \rceil) \\
&\|_{\mathsf{nf}}\ (\Diamond\ \neg\ R^{act}_{\leq n} \wedge (R^{act}_{\leq n} \xrightarrow{+} G_{\leq n})), \Box\ R^{act}_{\leq n+1} \\
\vdash\ &\Box\ (\Box\ U(S', S'') \rightarrow \Diamond\ P_{\leq n+1})
\end{aligned}$$

Before we sketch the proof of Lemma (7), we give its intuitive correctness argument: If component 2 is scheduled sufficiently often, then the violation of $R^{act}_{\leq n}$ occurs globally and leads to a contradiction since the environment of component 2 never violates it. Otherwise, the remaining component 1 ensures global progress.

Proof The symmetric versions of fairness rules (2.9) and (2.7) are used to get suitable induction hypotheses. The application of the latter fairness rule yields two cases. If component 2 is never scheduled again, then the remaining component 1 ensures global progress whenever $\Box\ U(S', S'')$ holds since P_{n+1} implies $P_{\leq n+1}$. Otherwise, a symbolic execution step of the interleaved components yields four types of proof obligations, depending on which component is scheduled and whether $R_{n+1}/R^{act}_{\leq n}$ holds or not. If a component is scheduled and its environment transition does not violate $R_{n+1}/R^{act}_{\leq n}$, then the case follows with induction. If a component is scheduled and its environment transition violates $R_{n+1}/R^{act}_{\leq n}$, then further symbolic execution is necessary to discard the goal, similarly to case 2 in the proof of RG rule (5.1). $\Box$

The final lemma considers the case where the rely conditions of both interleaved components are eventually violated locally. Such a violation, however, never occurs globally because each component (and the global environment) sustains the other component's rely. Since no safety formula exists for induction, fairness rules (2.7) and (2.9) for unfair interleaving are used instead to derive a contradiction.

Lemma 8 (Rely Eventually Violated in Components 1 and 2)
With the preconditions of RG rule 6.1, the following holds:

$$\begin{aligned}
&(\Diamond\ \neg\ R^{act}_{n+1} \wedge (R^{act}_{n+1} \xrightarrow{+} G_{n+1}) \wedge \lceil Actf_{=n+1} \rceil) \\
&\|_{\mathsf{nf}}\ (\Diamond\ \neg\ R^{act}_{\leq n} \wedge (R^{act}_{\leq n} \xrightarrow{+} G_{\leq n}) \wedge \lceil Actf_{\leq m} \rceil), \Box\ R^{act}_{\leq n+1} \\
\vdash\ &\mathit{false}
\end{aligned}$$

Proof Fairness rule (2.9) and its symmetric version are used to get two counters that count the steps until $\neg\ R^{act}_{n+1}$ and $\neg\ R^{act}_{\leq n}$ hold, respectively. The induction term is the sum of both introduced counters. Symbolically executing the interleaved formulas gives four types of proof obligations, similarly to the proof of Lemma 7, which are discarded by applying induction or further symbolic execution. $\Box$

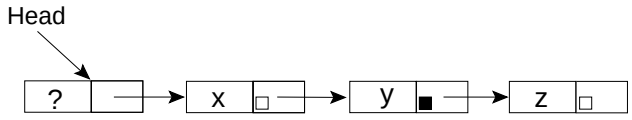
11.4. Case Study: A Lock-Free Set

This section illustrates the application of our compositional proof method for lock-freedom using a highly concurrent set implementation [70] that is part of Java's concurrency package [49]. In particular, the case study requires a non-trivial instance of the unchanged relation U. The focus is on the liveness aspects of the algorithm and we give only informal arguments for its safety. (For our complete proof of linearizability for the set data structure using our state-local refinement-based proof method from Section 10.1, see [59, 81]).

11.4.1. Michael's Lock-Free Set

The algorithm implements an abstract set of elements $x, y, \ldots$ using a singly-linked list of nodes: A node consists of an element (with selector .el) and a marked reference of type *mref* that is a pair of a boolean flag and a reference (of type *ref*) with selectors .marked and .nxt, respectively. We call a node unmarked/marked when its boolean flag is false/true. *Atomically* setting the mark of a node from false to true constitutes the logical deletion of its element from the set, as we explain below.

The nodes in the list are accessed from a shared variable *Head* that points to an unmarked dummy node. The dummy node helps to avoid special cases in the implementation. Nodes following the dummy node are ordered in strictly ascending order w.r.t. their elements. For instance, the singly-linked list



represents the set $\{x, z\}$ since x and z are unmarked and node y is marked. (An unfilled/filled rectangle denotes an unmarked/marked node.) Once a node is marked it can be physically deleted from the list in an extra step that changes the next reference, here the next pointer of node x is set to z. A physical deletion step does not have to be executed by the process that has marked a node, but can also be done by a concurrent process. This mutual helping scheme leads to a highly concurrent implementation.

The set offers methods `INSERT`(x), `DELETE`(x) and `LOOKUP`(x) for a given element x. All three methods crucially rely on an internal method `FIND`(x) that returns true if and only if it finds a node with element x in the list. Method `FIND`(x) ensures to take a snapshot of the list during its execution using three process local variables *Prev*, *Curr* and *Next* as Figure 11.1 shows: The pointer *Prev* points to an unmarked node with next reference *Curr*. If *Curr* is the null pointer, then all keys in the list must have been smaller than x when the snapshot was taken. Otherwise, *Curr* points to an unmarked node with an element that is greater or equal to x and a next reference *Next*. This constitutes an accurate snapshot of the `FIND` method.

We first explain the three data structure methods and then describe the internal method `FIND` in the following. Figure 11.2 shows these methods in pseudo code: Method `INSERT`(x) repeatedly calls method `FIND`(x) in line I2: If this call returns

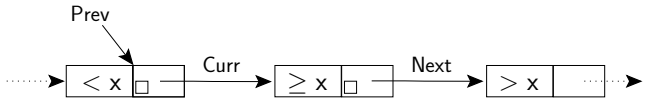
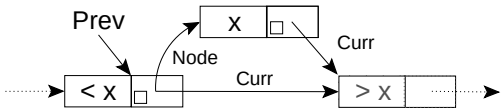


Figure 11.1.: An Accurate Snapshot of Method FIND.

```
node: (.el: elem, .mref: mref);
mref: (.marked: bool, .nxt: ref(node));
var Head: ref(mref);
lvar Prev: ref(mref);
lvar Curr, Next: ref(node);

I0 INSERT(x: elem): bool
I1   while true {
I2     if FIND(x)
I3       return false;
I4     var Node := alloc(node);
I5     Node.el := x;
I6     Node.mref := (false, Curr.nxt);
I7     if CAS(Prev, (false, Node), (false, Curr.nxt))
I8       return true;
I9   }

D0 DELETE(x: elem): bool
D1   while true {
D2     if ! FIND(x)
D3       return false;
D4     if ! CAS(ref(Curr.mref), (true, Next), (false, Next))
D5       continue;
D6     CAS(Prev, (false, Next), (false, Curr));
D7     return true;
D8   }

L0 LOOKUP(x: elem): bool
L1   return FIND(x)
```

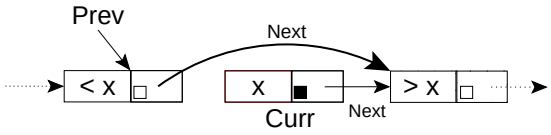
Figure 11.2.: The Lock-Free Set Methods INSERT, DELETE and LOOKUP.

true, then x is already in the set and the insert method fails, i.e., it returns false. Otherwise, the method tries to insert a new node between the previous and current node using the CAS instruction in line I7. It this CAS succeeds



then `INSERT` returns true, otherwise it retries the insertion of x by executing `FIND`(x) again.

Similarly, method `DELETE`(x) (repeatedly) calls method `FIND`(x). If this call fails, then the delete method fails as well and returns false since it has not found x in the set. Otherwise, it tries to logically delete the current node *Curr* using CAS in line D4. If this logical deletion fails, then the find method is retried. If it succeeds, then a physical removal of the marked node x is tried in line D6 using CAS



This step might fail if some other process helps by removing the marked node itself. In either case the method returns with true.

Finally, method `LOOKUP`(x) simply returns the value of method `FIND`(x). Its implementation is given in Figure 11.3. The basic idea of `FIND`(x) is to traverse the list using a previous and current pointer until it finds a current node with an element greater or equal x. Whenever the method encounters a marked current node during its traversal, it tries to physically delete it. If this deletion fails it restarts the traversal from the head of the list; otherwise, it continues its traversal.

More precisely, first a snapshot of the dummy node is taken (line F2) which serves as the initial previous node. Afterwards, the algorithm reads the marked reference in *Prev* and starts to traverse the list: If it finds the end of the list in line F5, it returns false since x was not found in the set. Otherwise, it reads the current node (lines F7, F8) and validates the snapshot of the previous node (line F9).[6] If this validation fails, then a retry from the head of the list is performed. Otherwise, the mark of the current node is checked (line F11). If the current node is marked, the method tries to physically remove this node from the list (line F17). If this physical removal fails, then the method retries from the head of the list. If the removal succeeds, the method updates its snapshot of the previous node (line F19) since it has been changed by the CAS and continues the list traversal. In case that the current node is not marked in line F11, the algorithm compares the element of the current node with x and either moves to the next node (line F14) or returns whether x is in the list (line F13).

An Informal Argument for Lock-Freedom. Informally, a CAS-based implementation is lock-free if it ensures that whenever one of its methods retries a CAS-loop,

[6] We assume that the algorithm runs under garbage collection to avoid an ABA problem, see Section 12.1 for more details on the ABA problem for lock-free algorithms.

```
F0  FIND(x: elem): bool
F1    retry:
F2      Prev := Head;
F3      (Pmark, Curr) := Prev;
F4      while true {
F5        if Curr = null
F6          return false;
F7        (Cmark, Next) := Curr.mref;
F8        val C_x := Curr.el;
F9        if Prev != (false, Curr)
F10         goto retry;
F11       if ! Cmark {
F12         if C_x >= x
F13           return C_x = x;
F14         Prev := ref(Curr.mref);
F15       }
F16       else /* Try physical deletion of the marked current node. */
F17         if ! CAS(Prev, (false, Next), (false, Curr))
F18           goto retry;
F19       (Pmark, Curr) := (Cmark, Next);
F20     }
```

Figure 11.3.: Method `FIND`.

some concurrent method has executed a step after which it can terminate. The set methods conform to this informal argument as follows:

When method `INSERT` must retry its loop due to a failing CAS in line I7, then the previous pointer *Prev* has been marked or its next reference has been modified. In the first case, the concurrent logical deletion of *Prev* leads to termination of the delete process that has marked the previous node. In the latter case where *Prev*'s next reference has been changed, two subcases must be discerned: 1) either a concurrent insert method has successfully added a new node right after *Prev*, but then this step leads to termination of the concurrent insert. 2) A concurrent method has physically removed *Prev*'s successor node from the list. Hence the length of the list has decreased (unless a concurrent insert has occurred meanwhile, but then this insert will terminate) and we can conclude with an inductive argument as the size of the list decreases in the absence of concurrent insertions.

Similarly, we can argue that whenever method `DELETE` must retry its loop (i.e., the CAS in line D4 fails), the concurrent modification of the current node *Curr* leads to progress of some other process. Thus it remains to consider the `FIND` method. If the validation of the previous pointer *Prev* fails in lines F9/F17, then the method must retry its outer loop, i.e., it restarts the list traversal. If these validation tests fail due to concurrent logical deletion/insertion, then some concurrent step leads to termination.

In case of a concurrent physical deletion of *Prev*'s next reference *Curr*, we can conclude progress with an inductive argument. Finally, the inner loop of `FIND` is only retried when the method moves to/physically deletes *Prev*'s next node. Both steps lead to the termination of the method itself, unless a concurrent insert occurs, but then again the insert ensures progress.

11.4.2. Termination Conditions for the Set

In this section we describe the instantiation of the unchanged predicate U for the set. According to our intuitive considerations above, we must consider the logical/physical deletion and insertion of nodes in the definition of U. Thus the instance of the unchanged relation for the set becomes the conjunction of three formulas *LDEL*, *PDEL* and *INS*

$$U(Head_0, H_0, Head_1, H_1) \equiv LDEL \wedge PDEL \wedge INS$$

where the global state for the set is simply the tuple $Head, H$ for the head pointer and the application's heap. (Due to the dual use of predicate U in both program and environment transitions, we use indices instead of priming for its parameters and denote the global state before/after a transition as $Head_0, H_0$ and $Head_1, H_1$, respectively.)

LDEL: The concurrent logical deletion of a node can invalidate the snapshot of a process and thus force it to retry a loop. Hence, the unchanged predicate precludes the concurrent logical deletion of nodes. Technically, this can be enforced by requiring that each unmarked list node remains in the list as an unmarked node

$$\forall\, r.\ reach(Head_0, r, H_0) \wedge \neg\, H_0[r].marked \rightarrow reach(Head_1, r, H_1) \wedge \neg\, H_1[r].marked$$

where predicate *reach(Head, r, H)* states that r is reachable from *Head*.

PDEL: The unchanged relation tolerates physical deletion, i.e., the number of marked nodes is allowed to shrink and thus the length of the rest list from any unmarked position in the list is allowed to decrease

$$\begin{aligned} &\#mnodes(Head_1, H_1) \leq \#mnodes(Head_0, H_0) \\ \wedge\ &(\forall\, r.\ reach(Head_0, r, H_0) \wedge \neg\, H_0[r].marked \rightarrow \#list(r, H_1) \leq \#list(r, H_0)) \end{aligned}$$

where $\#mnodes(Head, H)$ is the number of marked nodes reachable from the head of the list and $\#list(r, H)$ is the length of the rest list from reference r.

INS: To exclude concurrent insertion that can lead to a retry of a CAS-loop, we state that when the number of marked nodes is unchanged (hence no physical deletion occurs), then the next reference of each list node must stay the same and thus no insertion occurs either

$$\begin{aligned} &\#mnodes(Head_0, H_0) = \#mnodes(Head_1, H_1) \\ \rightarrow\ &(\forall\, r.\ reach(Head_0, r, H_0) \rightarrow reach(Head_1, r, H_1) \wedge H_0[r].mref = H_1[r].mref) \end{aligned}$$

11.4.3. Proving Termination For Each Individual Method

It is easy to see that our definition of predicate U is reflexive and transitive as required by the proof method. Thus it remains to prove the two state-local proof obligations (11.4) and (11.5) for each individual set method.[7] We consider (11.5) first which requires to show that a violation of predicate U in a transition of a method occurs only a finite number of times. Here we show the stronger property that after *one* violation of U each set method eventually terminates. Proofs are by symbolic execution and induction as explained in Sections 2.1.5 and 2.2.

Termination after Interference in Own Step. The proof of (11.5) for the set follows with the following simple observation: No step of method `FIND` ever violates the unchanged relation U

$$\texttt{FIND}(x; LS, S), \Box\ RI \vdash \Box\ U(S, S') \tag{11.10}$$

Proof The only step of `FIND` that modifies the global state is the CAS that physically removes the current node. However, this step does not violate U. □

Consequently, the only steps that violate the unchanged relation are either insertion or logical deletion of a node in lines I7/D4 after which the insert/delete method terminates. Hence, each set method $\texttt{COP}_p$ satisfies

$$\texttt{COP}_p(x; LS, S), \Box\ RI \vdash \Box\ (\neg\ U(S, S') \rightarrow \Diamond\ \textbf{last})$$

Proof The proof uses the compositionality rule (1.1) to replace the calls to find in each method using Lemma (11.10). Then it concludes with induction over the always formula in the succedent (2.5) and simple symbolic execution of sequential programs. □

This concludes the proof of proof obligation (11.5) for the set.

Termination without Interference from the Environment. The second proof obligation (11.4) is slightly more difficult to show. It requires to prove that each individual method terminates if it eventually runs without critical interference from its environment. The central part of the proof is to show that method FIND terminates if the unchanged predicate is eventually never concurrently violated:

$$\texttt{FIND}(x; LS, S), \Box\ RI \vdash \Diamond\ \Box\ U(S', S'') \rightarrow \Diamond\ \textbf{last} \tag{11.11}$$

Proof The proof is based on the following sublemma for the inner loop of the find method

$$\texttt{FIND}_{\text{inner}}, \Box\ RI, \Box\ U(S, S'), \Box\ U(S', S'') \vdash \Diamond\ \textbf{last} \tag{11.12}$$

which ensures termination of the inner loop of find under the assumption that predicate U is never violated (neither in program nor in environment transitions). The proof

[7] The local state vector LS is simply $Prev, Curr$ and $Next$; additional flags that are required in the abstract programming language to deal with loops are omitted, as well as rely and invariant conditions RI. These are given at [59, 81] where we also provide a proof of linearizability for the set based on our state-local proof obligations for linearizability (10.6).

of Sublemma (11.12) is straight-forward; it uses well-founded induction (2.3) on the length of the list from the previous pointer *Prev* onwards and symbolic execution of sequential programs.

Next, we lift Sublemma (11.12) to a similar lemma for the outer loop of `FIND`.

$$\texttt{FIND}_{\text{outer}}, \Box\ RI, \Box\ U(S, S'), \Box\ U(S', S'') \vdash \Diamond\ \textbf{last} \tag{11.13}$$

Finally, the proof of Lemma (11.11) introduces a counter for (lazy) induction using the liveness property $\Diamond\ \Box\ U(S', S'')$. A symbolic execution step then discerns whether $\Box\ U(S', S'')$ holds now. If this is the case, we execute the `FIND` method to reach its inner/outer loop and apply sublemmas (11.12)/(11.13) to discard the goal. Otherwise, we continue with symbolic execution and apply an induction hypothesis if a goal is repeated. This concludes the proof of Lemma (11.11). □

With Lemma (11.11), the proof of (11.4) for the lookup method is trivial. For the insert/delete methods, we must also take into account that when environment transitions never violate U, then the find method either terminates with an accurate previous/current snapshot or the number of marked nodes decreases due to a concurrent physical removal. With this additional property it is not difficult to show that the insert/delete methods also satisfy (11.4). Together, Theorem 11.1 then ensures that Michael's set implementation is lock-free. Full proof details are available at [59].

11.5. Summary

In this chapter, we have introduced a general compositional proof method for lock-freedom, including state-local versions of our proof obligations. Our method reduces the proof of lock-freedom to simple process-local termination proofs for sequential programs based on suitable termination conditions. These process-local proofs are simple, since they do not have to deal with possible crashes of a process and environment transitions terminate in each step of symbolic execution. Different from other approaches which only argue informally that their decomposition is correct, we provide a mechanized soundness proof for our method in KIV.

Furthermore, we have illustrated the application of our state-local proof obligations to prove lock-freedom for a highly concurrent set implementation. This proof is also fully mechanized in KIV. To our knowledge, the only place in the literature where the verification of the set algorithm is briefly mentioned is [38]. For an extended comparison see Chapter 13.

In the next chapter we deal with the important issue of verifying lock-free data structures in environments without (lock-free) garbage collection where specific lock-free memory management techniques are required.

12. Further Case Studies: Lock-Free Memory Reclamation

In this chapter, we apply RGITL to verify the correctness (linearizability and lock-freedom) of lock-free data structures in environments without garbage collection. In such a setting, lock-free data structures must be extended with custom lock-free memory management to avoid memory leaks. A naive approach of freeing memory typically leads to memory access faults. The direct reuse of memory locations leads to the notorious ABA problem of these algorithms [102] where a data structure can be corrupted when a memory location is reinserted with modified contents and this reinsertion is not detected by a CAS instruction. In particular, we study extended data structures that use the common lock-free memory reclamation techniques of modification counters [102] and hazard pointers [68], respectively. We have mechanized all proofs in KIV [58] and to our knowledge, both examples had no (mechanized) verification before. The exposition in Section 12.1 widely corresponds to our article [95]. Section 12.2 closely follows our article [96].

The rest of this chapter is structured as follows: In Section 12.1, we consider two lock-free stacks with modification counters: To deal with the heap and to avoid reasoning about the local states of other processes, we use an approach that combines ownership annotations and separation logic. In Section 12.2, we consider the more advanced technique of hazard pointers: We exploit the symmetry of the hazard pointers technique and apply our state-local RG rule for the verification.

12.1. Two Stacks with Modification Counters

In this section, we consider the verification of two lock-free stacks with modification counters. The implementation is based on the well-known lock-free Treiber stack [102]. It uses generic lock-free push and pop operations in two contexts: First, to add/remove *elements* from a lock-free stack and second, to add/remove *heap locations* from another stack that serves as a lock-free memory allocator. While several other proofs for the basic stack exist, these have mainly focussed on linearizability for the simple version under garbage collection (GC). Here we consider linearizability and lock-freedom of a version that is close to a real implementation in an environment without GC.

The verification of our version poses additional challenges w.r.t. reasoning about the heap, the fundamental ABA problem of lock-free algorithms and compositional verification (when verifying the client code, we want to reuse the proofs of the generic stack operations). Moreover, it gives an example of our proof method for lock-freedom

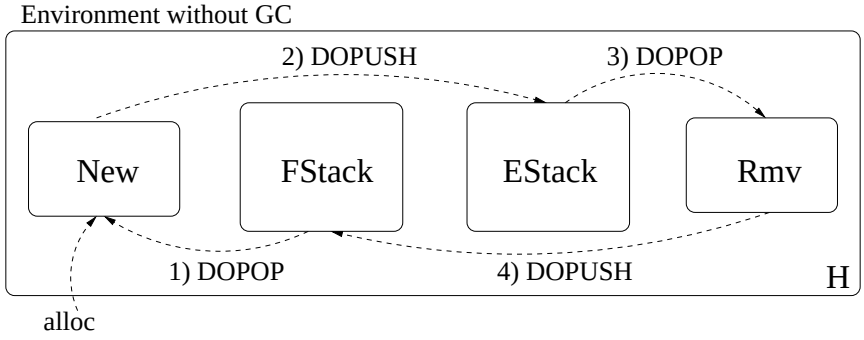


Figure 12.1.: Memory Reuse for Element/Free Stack.

where the unchanged relation can be violated several times in steps of a method before termination (see proof obligation (11.3)).

12.1.1. The Stack Algorithms

The core data structure is a lock-free stack (EStack) that is used by a finite number of concurrent processes to store arbitrary elements. Since the application's environment does not offer GC, a second lock-free stack is used to allocate and free memory and thus to avoid memory leaks. This stack is called the free stack (FStack) in the following. Both stacks are implemented as singly linked lists of nodes (pairs of values and locations having *.el* and *.nxt* selector functions). A shared variable *ETop* marks the top node of the element stack; it is a pair of a node reference and a modification counter (of type *nat*), with selector functions *.ref* and *.cnt*, respectively. Similarly, the free stack is accessible from a shared variable *FTop*.

As shown in Figure 12.1, each process that wants to push an element on the stack, first allocates new memory (NEW) from the free stack 1) and resorts to the machine's allocator (alloc) only if the free stack is currently empty. 2) The new node is pushed on the element stack. 3) Whenever a process removes a node from the element stack (Rmv), after reading its element, it pushes 4) the node on the free stack, thus making its memory available for (concurrent) reuse.

To detect this concurrent reuse, a modification counter is added to each of the two ABA prone top-of-stack pointers. This counter is incremented atomically with (either) the insertion or removal of a location from the data structure, thus making memory reuse visible to CAS which now compares both a location and a modification counter. The theoretical chance of bogus behavior due to wrap around of a modification counter is negligible [102] and therefore, we model modification counters as natural numbers.

Figure 12.2 shows the concrete algorithms in pseudo code. The client procedures `PUSH` and `POP` use two generic procedures `DOPUSH` and `DOPOP`, which operate on either the element or the free stack. Operation `DOPUSH` repeatedly tries to switch a shared top-of-stack pointer *Top.ref* to a new node using CAS in a lock-free manner. It repeatedly

```
node = (.el: elem, .nxt: ref(node));
rc   = (.ref: ref(node), .cnt: nat);
var ETop, FTop: ref(rc);

PUSH(a: elem)
  var new := DOPOP(FTop);
  if (new = null) {
    alloc(new);
  };
  new.el := a;
  DOPUSH(new, ETop)

POP(): empty | elem {
  var rtop := DOPOP(ETop);
  if (rtop = null) {
    return empty;
  };
  var lout := rtop.el;
  DOPUSH(rtop, FTop);
  return lout

DOPOP(Top: ref(rc)): ref(Node)
  var ltop: rc;
  repeat
    ltop := Top;
    if (ltop.ref = null) {
      break
    };
    nxt := (ltop.ref).nxt;
  until
    CAS(Top, (nxt, ltop.cnt), ltop);
  return ltop.ref


DOPUSH(node: ref(Node), Top: ref(rc))
  var ltop: rc;
  repeat
    ltop := Top;
    node.nxt := ltop.ref;
  until
    CAS(Top, (node, ltop.cnt + 1), ltop)
```

Figure 12.2.: Lock-Free Element and Free Stack Operations.

takes an atomic snapshot of *Top* including both the current top-of-stack pointer and its modification counter. After setting the new node's next pointer to the snapshot's location, the new node becomes the target of a subsequent CAS: if it succeeds, the node is atomically added to the stack and the modification counter is incremented. The generic pop operation `DOPOP` works similarly. It repeatedly reads the shared top: If its pointer is *null*, then it immediately returns *null*. Otherwise, its next reference becomes the target of a subsequent CAS. If this CAS succeeds, the top node is removed from the stack and the snapshot's location is subsequently returned.

Without a modification counter, an ABA problem could occur as follows: suppose that a pop process p takes a snapshot of the top pointer when the element stack consists of exactly one node at location A and the free stack is empty. Process p is preempted after setting its local reference *nxt* to null for another process, which removes A from the element stack without yet freeing it. Subsequently, a third process q executes a successful push, thereby allocating a new location B (by resorting to the machine's allocator). Then A is freed and q pushes A on the element stack, which now has two nodes at locations [A, B]. If now p is rescheduled, its CAS would erroneously succeed, removing both nodes A and B at once and possibly returning an unexpected value.

12.1.2. Challenges of Verifying the Two Stacks

Compared to the simple version of the stack under GC, additional verification challenges arise from having an explicit memory allocator: First, it becomes necessary to prove that the application does not leak memory. Here we use ownership annotations and separation logic's star operator to state that the application's heap is always separated into three distinct parts: One for the element stack, another one for the free stack and a third part that is owned by some of the (running) processes. Second, we have to show that an ABA problem does not occur. Here RG reasoning permits to express ABA prevention as an appropriate rely condition. Third, to avoid big redundant proofs, we want to verify the generic procedures `DOPUSH` and `DOPOP` separately, and reuse these proofs as contracts in the verification of the sequential client code `PUSH` and `POP`, respectively. Here the sequential compositionality of RGITL (see rule (1.1)) is crucial, which allows us to replace procedure calls with appropriate linearizability/lock-freedom abstractions. Finally, since individual processes might now starve while accessing either the element or the free stack, the argument why individual starvation does not lead to a global livelock becomes non-trivial. Yet the proof of lock-freedom using our compositional proof method is straightforward.

From a practical point of view, further techniques could be added to the implementation to make it more efficient. E.g., when the contention on the top-of-stack pointers is high, elimination [46] could be used to pair-off concurrent push and pop operations without accessing the stack data structure. Of course, this would lead to more complex linearization arguments that can be handled by our generic proof method for linearizability (from Section 9.3). Without elimination, the linearization points for the case study are not difficult and we can use our refinement-based proof method for linearizability instead (see Section 10.1).

12.1.3. The Concrete Stack Specification with Ownership

This section gives the specification of the stack algorithms in RGITL's abstract programming language. It turns out to be beneficial to use ownership annotations here as well, similar to our previous use of ownership for the multiset with fine-grained locking in Section 10.2. Different from the multiset where we have used ownership for array slots, we now use ownership for memory locations of a concurrent heap.

A concurrent heap with ownership is simply a partial function

$$H : \mathit{ref} \rightharpoonup (\mathit{node}, \mathit{owner})$$

from locations to nodes with owner o.[1] In our case study, it is sufficient to discern three possible owners per heap location:

$$\mathit{owner} \ ::= \ p \mid \mathit{estack} \mid \mathit{fstack}$$

That is, each heap location r is either owned by some process p, i.e., $H(r).\mathit{owr} = p$, or it belongs to either the element or the free stack.

Figure 12.3 shows the program specification that takes ownership into account. The effects of these simple auxiliary state annotations are again worth noting: First of all, no further heap disjointness properties must be defined, since they are already implied by the ownership annotations. (Our technical report [97] shows that several disjointness properties between two local states would be necessary without ownership annotations, using our state-local proof obligation for linearizability (10.6).) Second, we can completely avoid talking about local variables, in particular program labels. Hence, when applying rule 6.1, the state variables S can be simply instantiated with the tuple $\mathit{ETop}, \mathit{FTop}, H$, which is the shared state of the algorithm where interference can actually occur. Third, we can uniformly handle typical heap modifications and use separation logic on (owned) heap predicates to avoid inductive reachability arguments. This is further explained in the next subsection.

12.1.4. Concurrent Heaps with Ownership and Separation

Instead of integrating heaps into the semantics of RGITL, we use a lightweight embedding of separation logic into higher-order logic (available as a library in KIV [9]), where heap assertions are encoded as heap predicates P, Q of type $\mathit{heap} \to \mathit{bool}$. The lifting of this theory to heaps with ownership is done in the standard way: an owned heap predicate $o[P]$ with owner o holds over heap H, iff P holds and every location in H has owner o. Similarly, the common operators from separation logic are overloaded. For instance, the star operator between two owned heap predicates $o_0[P]$ and $o_1[Q]$ now has the following semantics:

$$\begin{aligned}(o_0[P] \ \mathbf{*} \ o_1[Q])(H) \equiv \exists\ & H_0, H_1. \\ & \mathit{dom}(H_0) \cap \mathit{dom}(H_1) = \emptyset \wedge (H_0 \ \cup \ H_1 = H) \wedge P(H_0) \wedge Q(H_1) \\ \wedge \forall\ & r.\ (r \in H_0 \to H_0(r).\mathit{owr} = o_0) \wedge (r \in H_1 \to H_1(r).\mathit{owr} = o_1)\end{aligned}$$

[1] In KIV, the heap H is actually represented as a tuple (D, Nf, Of) with $\mathit{dom}(H) = \mathrm{D}$, node function Nf and an auxiliary ownership function Of.

```
PUSH_p(a; ETop, FTop, H) {
  let New in {
   DOPOP_p(FTop, H, New);
   if New = null {
     choose New_0 with (New_0 ≠ null ∧ New_0 ∉ dom(H)) in {
       dom(H) := dom(H) ∪ {New_0}, New := New_0, H(New_0).owr := p
     }};
   H(New).el := a;
   DOPUSH(estack, New; ETop, H)}}

POP_p(ETop, FTop, H, Out) {
  let LOut = empty, ROut in {
   DOPOP_p(ETop, H, ROut);
   if ROut ≠ null {
     LOut := H(ROut).el;
     DOPUSH(fstack, ROut; FTop, H);
   };
   Out := LOut}}

DOPUSH(o, Node; Top, H) {
  let Succ = false, LTop in {
   while ¬ Succ {
     LTop := Top;
     H(Node).nxt := LTop.ref;
     if* LTop = Top {   /* CAS */
       Top := (Node, LTop.cnt + 1), Succ := true, H(Node).owr := o;
     }}};

DOPOP_p(Top, H, ROut) {
  let Succ = false, LTop, Nxt in {
   while ¬ Succ {
     LTop := Top;
     if LTop.ref = null { Succ := true }
     else {
       Nxt := H(LTop.ref).nxt;
       if* LTop = Top {   /* CAS */
         Top.ref := Nxt, Succ := true, H(LTop.ref).owr := p;
       } }};
   ROut := LTop.ref}}
```

Figure 12.3.: RGITL Specification of Stacks with Ownership (Shaded).

In a concurrent setting, assertions about the permissions of processes to access shared resources are typically required. Again, we do not enrich the semantics of RGITL with permissions, but simply define them based on ownership. It is common to assume that a heap location, which is owned by some *process* can only be read by others, but neither deallocated, nor modified. The following "permission rely" encodes this restriction.

$$\begin{aligned} PR_p(H', H'') \equiv \forall\ r.\ & \\ & ((H'(r).owr = p \wedge r \in H') \leftrightarrow (H''(r).owr = p \wedge r \in H'')) \\ \wedge\ & (H'(r).owr = p \wedge r \in H' \rightarrow H'(r) = H''(r)) \end{aligned}$$

The rely $PR_p(H', H'')$ implies the following stability property

$$(p[P]\ *\ true)(H') \wedge PR_p(H', H'') \rightarrow (p[P]\ *\ true)(H'')$$

and it is easy to prove that under PR_p, none of the annotated operations changes any portion of the heap, which is owned by another process.

To express absence of memory leaks, three simple heap predicates are defined: predicate $owned(H)$ states that each r in H is owned by some process; $owns0_p(H)$ denotes that p owns no location in H, and $owns1_p(r, H)$ denotes that p owns exactly location r in H. Obviously, predicates $owns0_p$ and $owns1_p$ are stable over the permission rely PR_p, and it is easy to show that predicate $owns0_p$ is an idle state pre- and post-condition of the annotated programs.

Finally, to verify linearizability we want to express that some abstract list x is represented by a heap location r. The heap predicate $lst(r)$ defines this property, and the * operator enforces acyclicity of the heap structure under r.

$$\begin{aligned} & lst(r) = \textbf{if}\ r = null\ \textbf{then}\ ls(r, []) \ \textbf{else}\ \exists\ a, x.\ ls(r, a + x) \\ & ls(r, []) = \text{emp} \wedge r = null \\ & ls(r, a + x) = \exists\ r_0.\ ((r \mapsto (a, r_0))\ *\ ls(r_0, x)) \end{aligned}$$

where emp holds for the empty heap only

$$\text{emp}(H) \leftrightarrow dom(H) = \emptyset$$

and $(r \mapsto (a, r_0))$ defines a heap consisting of one node at location r, which stores element a and a next reference r_0

$$(r \mapsto (a, r_0))(H) \leftrightarrow r \neq null \wedge dom(H) = \{r\} \wedge H(r).el = a \wedge H(r).ref = r_0$$

12.1.5. Proving Linearizability and Lock-Freedom for the Stacks

The proofs apply our refinement-based proof method for linearizability (Theorem 10.1) and our proof method for lock-freedom (Theorem 11.1). A state-local version is not necessary here, since specifications are over the shared state only. We start by defining the concrete instances of the RG parameters of these theorems, based on the previous

notions of concurrent heaps. Then we briefly sketch the compositional proofs of linearizability and lock-freedom for the stack which mainly apply symbolic execution of sequential programs (see Sections 2.1.5 and 4.3).

Instantiating the RG Predicates for the Stack. Procedure $\texttt{COP}_p$ executes either $\texttt{PUSH}_p$ or $\texttt{POP}_p$ here, depending on the given operation index. The state variable S is simply $ETop, FTop, H$. The global initial state condition *Init* requires H to be empty, and both $ETop$ and $FTop$ to be $(null, 0)$. The idle state predicate $Idle_p(ETop, FTop, H)$ is simply $owns0_p(H)$. Since each process owns no portion of H in idle states, the application does not leak memory.

The invariant $Inv_{estack,fstack}(ETop, FTop, H)$ claims that the heap always consists of two distinct linked lists with owner *estack*/*fstack* and a separate portion where each location is owned by some process

$$Inv_{o,o_0}(Top, Top_0, H) \equiv (o[lst(Top.ref)] \mathbin{*} o_0[lst(Top_0.ref)] \mathbin{*} owned)(H)$$

where $o, o_0 \in \{estack, fstack\}$ such that for $o = estack$ the corresponding top-of-stack pointer is $ETop$ and so forth.

For stack nodes with owner *estack* or *fstack*, the possible concurrent access is determined by the specific use of modification counters. In contrast to locations that are owned by some process, both the content and the ownership information of a stack location can change when the location is concurrently removed from the data structure. An appropriate stack rely condition that captures the correctness of the memory reclamation protocol and ensures that an ABA problem does not occur on neither the element nor the free stack is the following.

$$\begin{aligned}
&SR(o,\, Top', H',\, Top'', H'') \equiv \\
&\quad Top'.cnt \leq Top''.cnt \\
&\wedge\ (Top'.ref \neq null \rightarrow \\
&\qquad Top' = Top'' \wedge H'(Top'.ref) = H''(Top'.ref) \wedge H''(Top'.ref).owr = o \\
&\quad \vee\ H''(Top'.ref).owr \neq o \vee\ Top'.cnt < Top''.cnt) \\
&\wedge\ (\forall\ r \neq null.\ H'(r).owr \neq o \rightarrow H''(r).owr \neq o \vee\ Top'.cnt < Top''.cnt)
\end{aligned}$$

The specific stack relies $SR(estack, \dots)$ and $SR(fstack, \dots)$ ensure that during $\texttt{DOPOP}$ on one of the two stacks, the ABA prone snapshot location either stays in the stack and its contents (including ownership annotation) are unchanged, or if it is concurrently removed, then it is not reinserted unless the modification counter is increased.

Finally, it remains to define the full rely R_p as the conjunction

$$\begin{aligned}
&R_p(ETop', FTop', H', ETop'', FTop'', H'') \equiv \\
&\quad PR_p(H', H'') \wedge SR(estack, ETop', H', ETop'', H'') \wedge SR(fstack, FTop', H', FTop'', H'')
\end{aligned}$$

and the guarantee as the conjunction of all other relies:

$$\begin{aligned}
&G_p(ETop, FTop, H, ETop', FTop', H') \equiv \\
&\quad \forall\ q \neq p.\ R_q(ETop, FTop, H, ETop', FTop', H')
\end{aligned}$$

Verifying RG Assertions for the Stacks. Our refinement-based proof method for linearizability, Theorem 10.1, requires to prove RG assertion (10.1) and the refinement (10.2) for each individual data structure operation. First we sketch the proof of the RG assertion for the sequential code of PUSH and POP.

By splitting RG assertions for sequential programs, two main subgoals evolve for DOPUSH/DOPOP that we prove with the following two sublemmas

$$\begin{array}{l} \quad \mathit{owns1}_p(\mathit{New}, H), \mathit{Inv}_{o,o_0}, \Box\ \mathit{New}' = \mathit{New}'' \\ \vdash [\mathit{PR}_p \wedge \mathit{SR}_o \wedge \mathit{SR}_{o_0}, (\forall\ q \neq p.\ \mathit{PR}_q) \wedge \mathit{SR}_o \wedge \mathit{SR}_{o_0}, \mathit{Inv}_{o,o_0}, \texttt{DOPUSH}(o, \dots)] \\ \quad \mathit{owns0}_p(H) \end{array}$$

and

$$\begin{array}{l} \quad \mathit{owns0}_p(H), \mathit{Inv}_{o,o_0}, \Box\ \mathit{New}' = \mathit{New}'' \\ \vdash [\mathit{PR}_p \wedge \mathit{SR}_o \wedge \mathit{SR}_{o_0}, (\forall\ q \neq p.\ \mathit{PR}_q) \wedge \mathit{SR}_o \wedge \mathit{SR}_{o_0}, \mathit{Inv}_{o,o_0}, \texttt{DOPOP}_p] \\ \quad \textbf{if}\ \mathit{New} = \mathit{null}\ \textbf{then}\ \mathit{owns0}_p(H)\ \textbf{else}\ \mathit{owns1}_p(\mathit{New}, H) \end{array}$$

The proof of the sublemmas uses induction rule (4.3) and symbolic execution for sequential programs according to Section 4.3. In the proofs, the current shape of the heap is typically given by the invariant Inv_{o,o_0} above and a formula $(p[(\mathit{Ref} \mapsto \mathit{node})]$ * $\mathit{true})(H)$, which defines the local state of the current process (*Ref* corresponds to either the local variable *New* or *LTop.ref*). To transfer local state to and from one of the stacks, two simple split/merge lemmas are used for the push/pop algorithms.

$$(p[P] * R)(H) \wedge (o[P] * o_0[Q] * \mathit{owned})(H) \rightarrow (o[P] * o_0[Q] * p[P] * \mathit{owned})(H)$$
$$(p[P] * \mathit{owned} * R)(H) \rightarrow (\mathit{owned} * R)$$

Verifying DOPOP is most challenging, since transitive arguments over several symbolic execution steps are required to derive that if the snapshot location is concurrently removed, the following CAS operation does fail.

Verifying the Refinement for the Stack. Having verified the RG assertion (10.1), from the local view of one process executing a stack operation, all environment transitions preserve its rely at all times and the invariant can be assumed to hold in each state, i.e., $\Box\ \mathit{RI}$ holds locally. This property is now used in a local refinement proof according to (10.2) where the abstraction function for the element stack simply corresponds to formula $\mathit{estack}[\mathit{ls}(\mathit{ETop.ref}, \mathit{Stack})]$.

To be compositional, the refinement proofs use rule (1.1) to replace the calls to the client procedures with suitable abstractions. These abstractions state that the generic push and pop operations basically refine $\mathbf{skip}^*; \mathit{AOP}; \mathbf{skip}^*$ or just $\mathbf{skip}^*$ for the element and free stack, respectively. (Here *AOP* abbreviates the atomic relation that either pushes or pops an element from the abstract *Stack* of elements (8.1).) We give the concrete instances of these sublemmas for DOPUSH in the following. (The lemmas for DOPOP are similar).

A push on the element stack can be abstracted according to the following refinement lemma

$$(p[New \mapsto a \times r] \ast true)(H), \texttt{DOPUSH}(estack, New; ETop, \dots), \Box\ (RI \wedge New' = New'') \vdash \mathbf{skip}^*; Stack := a + Stack; \mathbf{skip}^*$$

Similarly, we use the following refinement lemma for a push on the free stack

$$owns1_p(Ref, H), \texttt{DOPUSH}(fstack, Ref; FTop, \dots), \Box\ RI \vdash \mathbf{skip}^*$$

where we have already passed the linearization point and thus the remaining abstract program executes non-blocking stutter steps only.

Verifying Lock-Freedom for the Stack. The unchanged relation U from our proof obligations for lock-freedom (11.2) and (11.3) is simply defined as the identity relation over $ETop$ and $FTop$. The actual proofs of lock-freedom are then compositional, i.e., they verify suitable sublemmas for the generic stack operations `DOPUSH` and `DOPOP`. These proofs are largely automatically. Then we apply rule (1.1) to replace the calls to these generic methods in the client code and the proof for the client code completes with simple symbolic execution steps according to Theorem 2.2. The two sublemmas that we use for the generic pop method `DOPOP` are

$$\texttt{DOPOP}_p(Top, H, ROut), \Box\ RI, \Box\ ROut' = ROut'' \vdash \Diamond\ \Box\ Top' = Top'' \rightarrow \Diamond\ \mathbf{last}$$

and

$$\texttt{DOPOP}_p(Top, H, ROut), \Box\ RI, \Box\ ROut' = ROut'' \vdash \Box\ \Diamond\ Top \neq Top' \rightarrow \Diamond\ \mathbf{last}$$

For the generic push method we use similar lemmas. This concludes our verification for the lock-free stacks with modification counters. For an online description of the KIV proofs see [58].

12.2. A Stack with Hazard Pointers

In the previous section we have seen a simple and common technique to compensate the absence of GC for lock-free data structures and to deal with the ABA problem. One major disadvantage of the approach with modification counters is that it does not permit memory deallocation, but only memory reuse within the same application. Moreover, it typically requires double-word CAS which is not as widely supported as single-word CAS. Hazard pointers [68] overcome these restrictions: They enable safe memory reclamation by extending concurrent data structures with their own wait-free garbage collection. In this section, we analyze the central correctness aspects of hazard pointers and apply the results to verify a lock-free stack with hazard pointers.

Proving safe memory reclamation and ABA-avoidance for a stack with hazard pointers is challenging [80]. To our knowledge, our proof is the first mechanized proof of such an algorithm. Different from other existing pen and paper proofs, our proof

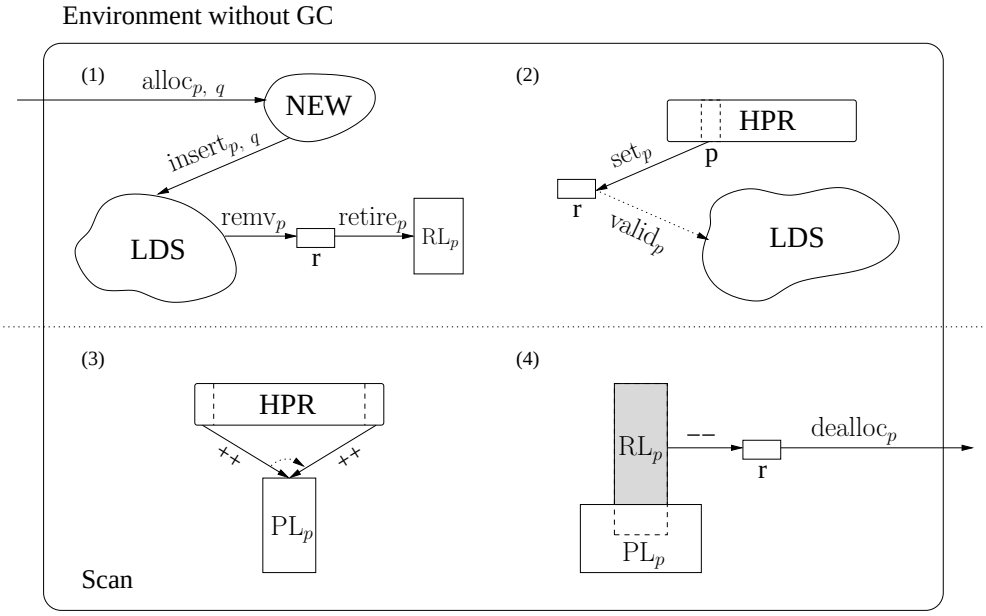


Figure 12.4.: Michael's Hazard Pointers Method.

considers all aspects of the reclamation scheme: Memory-safety, ABA-prevention as well as preservation of linearizability and lock-freedom of the stack. We use neither complex history variables [80] nor do we need to reason about the past [37] to capture the essential properties. Furthermore, we show that all verification conditions can be expressed in terms of at most two concurrent processes (see Section 12.2.5). This helps to keep the specifications/proofs moderately complex.

At the end of this section, we discuss two extensions of our work that we have also mechanically verified [59]: One non-trivial extension that to our knowledge has not been considered before in a formal proof, is the verification of the stack with hazard pointers that are written and read *non-atomically*. Furthermore, using the well-known lock-free MS queue [69], we briefly discuss how our approach can be transferred to verify other data structures with hazard pointers.

12.2.1. Hazard Pointers

Figure 12.4 illustrates the hazard pointers technique:

(1) processes $p, q, \ldots$ can concurrently allocate and insert new objects *NEW* to a lock-free data structure *LDS*. Every process p collects the memory of objects that it removes from *LDS* in a local pool of retired locations RL_p. These locations are potential candidates for deallocation. However, the concurrent use of these retired locations must be taken into account to ensure correctness.

(2) each process is associated with a fixed (small) number of single-writer multi-reader shared pointers, so called hazard pointers. All hazard pointers of all processes are contained in a global hazard pointer record *HPR*. By setting one of its hazard pointers to a location r, process p signals other contending processes not to deallocate this location. Crucially, to ensure that this signal is indeed considered, p subsequently checks whether r is still part of *LDS*. Only if this check called hazard pointer validation succeeds enters process p a subsequent code region that accesses r.

To deallocate memory, process p executes a scan method in two phases (3) and (4). In (3), it consecutively collects all hazard pointers of all processes in a local pointer list PL_p by traversing *HPR*. In (4), all retired memory locations r that were not found when traversing *HPR* (i.e., $r \in RL_p - PL_p$) are freed to the environment's memory management system for arbitrary reuse.

A properly extended lock-free data structure with hazard pointers has the following central correctness property:

A validated hazard pointer is not concurrently freed.	(12.1)

This is because at the time of its successful validation, a hazard pointer is in *LDS* and hence in no retired list. Consequently, no currently running scan will deallocate it. After its successful validation, a hazard pointer might be concurrently retired while still being used. Yet it is not freed, since the retiring process collects the pointer during its traversal of *HPR*.

12.2.2. The Lock-Free Stack with Hazard Pointers

Figure 12.5 gives the extended stack methods in pseudo code. The push method remains unchanged. Just as already shown in Figure 12.2, whenever a process executes a push, it first allocates a new node *New* and initializes it with input value *In*. Then it repeatedly tries to CAS the global top to point to this new node.

The pop method must be adjusted to ensure its correctness: It requires one hazard pointer to cover the critical usage of the snapshot pointer *ltop*. This hazard pointer is atomically set using the global hazard pointer array *HPR* at index p which is the identifier of the current process. Before accessing *ltop* again, the hazard pointer is validated. Crucially, only after this test succeeds, can it be guaranteed that the snapshot node is not concurrently deallocated and possibly reused. Finally, a location that has been removed from the stack is added to the local list *RL* of retired references of the current process. The scan routine is performed at the end of pop, depending on whether the current number of retired locations has exceeded a given threshold.

The scan method deallocates retired locations that are not concurrently used. In its first loop, a scan sequentially traverses the hazard pointer record: This includes atomically taking a snapshot *lhp* of the *HPR* entry at the current (process) index and adding it to a local pointer list *PL*. In the second loop, memory locations that have been retired by the current process but are not in *PL* are consecutively deallocated.

```
node = (.el: elem, .nxt: ref(node));
var Top: ref(node);
var HPR: array[0..MAXID] of ref(node);
lvar RL: list of ref(node);
const nat THRESHOLD > 0;

PUSH(e: elem)
  var ltop, node: ref(node);
  node := alloc(node);
  node.el := e;
  repeat
    ltop := Top;
    node.nxt := ltop;
  until
    CAS(Top, node, ltop)


SCAN()
  var PL := [];
  for (i := 0; i <= MAXID; i++) {
    lhp := HPR[i];
    if (lhp != null) {
      PL.insert(lhp)
    }
  };
  foreach r in (RL \ PL) {
    RL.delete(r);
    dealloc(r)
  }
```

```
POP(): empty | elem
  var ltop, nxt: ref(node);
  repeat
    ltop := Top;
    if (ltop = null) {
      return empty;
    };
    HPR[p] := ltop;
    if (ltop = Top) {
      nxt := ltop.nxt
    } else { continue };
  until CAS(Top, nxt, ltop);
  var el := ltop.el;
  RL.insert(ltop);
  if (RL.size > THRESHOLD) {
    SCAN();
  };
  return el
```

Figure 12.5.: Pseudo Code of the Stack with Hazard Pointers.

12.2.3. The Concrete Specification of the Stack with Hazard Pointers

Figure 12.6 shows the specification of the extended stack algorithms in RGITL's abstract programming language. (Shaded instructions denote the required extensions of the stack algorithms/specification under GC.) To simplify verification while maintaining the core ideas of Michael's algorithm, our model uses several abstract data structures. In particular, we use a function $HPR\colon nat \to ref$ from naturals to locations to model the hazard pointer record, while Michael proposes a singly linked heap list. The local list of retired references is modeled as an algebraic list RL. The application's heap is simply a partial function from references $r\colon ref$ (with $null \in ref$) to nodes with standard algebraic methods, e.g., deallocation is written $H - r$. (We do not consider ownership annotations in the current specifications/proofs, but we discuss at the end of this section how they can be used for this case study.) Moreover, in the second loop of the scan method, the **choose** summarizes some merely local steps that are required to determine the references $RL - PL$ that can be safely deallocated. Furthermore, we allow a scan to be performed arbitrarily between stack methods, while Michael calls a scan at the end of pop, depending on the current number of retired locations. As a simple extension, we consider the possible reset of a hazard pointer `RESET` between executions of push, pop or scan, while the original code does not explicitly reset. As a more challenging extension, we have also verified this algorithm using *non-atomic* read and writes of hazard pointer entries, see below.

Finally, the identifier of the current process $Id\colon nat$ is part of the local state and we use a few simple boolean flags to mark code regions for the verification, since the logic does not use program counters: The boolean flag $Hazard_{pc}$ marks the critical code region in the pop method where the validated hazard pointer equals (covers) the snapshot $LTop$ and the flag $BefInc_{pc}$ discerns whether the current increment has already occurred during the traversal of HPR.

This section shows general properties of hazard pointers and their specialization to formal verification conditions for the stack from Figure 12.6. To keep the presentation readable, we only give the main conditions explicitly, all formal conditions are given at [59]. All conditions are expressed in terms of at most two processes. This is possible, since a retired location r can only be freed by the process which has removed r from the stack and then retired it. Thus, when a process is in its critical code region, there is at most one other process which could theoretically free its critical pointer. Hence, the example can be handled by our state-local proof obligations for linearizability (10.6) and lock-freedom (11.4)/(11.5).

12.2.4. General Properties of Hazard Pointers

In this section we define general properties of hazard pointers and explain their relationship with standard GC. The following central invariant property of hazard pointers ensures that heap access errors do not occur in critical code regions.

$$\boxed{HPR_{valid} \subseteq H} \tag{12.2}$$

```
PUSH(In; New, SuccPush, Top, H) {
  let LTop in {
    choose New_0 with (New_0 ≠ null ∧ New_0 ∉ dom(H)) in {
      New := New_0, dom(H) := dom(H) ∪ {New_0}, SuccPush := false
      H(New).el := In;
      while ¬ SuccPush do {
        LTop := Top;
        H(New).nxt := LTop;
        if* LTop = Top {   /* CAS */
          Top := New, SuccPush := true
        }; }}}

POP(; Id, Hazard_pc, LTop, OSucc, RL, Top, H, HPR, Out) {
  let Nxt, LOut = empty in {
    OSucc := false;
    while ¬ OSucc do {
      LTop := Top, Hazard_pc := false;
      if LTop = null then {
        OSucc := true
      } else {
        HPR(Id) := LTop;
        if* LTop = Top then {
          Hazard_pc := true;
          Nxt := H(LTop).nxt;
          if* LTop = Top {   /* CAS */
            Top := Nxt, OSucc := true
          }}}};
    if LTop ≠ null then {
      LOut := H(LTop).el';
      RL := LTop + RL, Hazard_pc := false}
    Out := LOut}}

SCAN(; Scan, BefInc_pc, Lid, Lhp, PL, RL, H, HPR) {
  PL := [], Scan := true;
  while Lid ≤ MAXID do {
    Lhp := HPR(Lid), BefInc_pc := true;
    if Lhp ≠ null then {
      PL := Lhp + PL}
    Lid := Lid + 1, BefInc_pc := false};
  while Scan do {
    choose Ref with (Ref ∈ RL − PL) in {
      RL := RL − Ref, H := H − Ref}
    ifnone Scan := false, Lid := 0}}

RESET(; Id, HPR) {HPR(Id) := null}
```

Figure 12.6.: RGITL Specification of Lock-Free Stack with Hazard Pointers.

That is, each validated hazard pointer is in the application's heap at all times, i.e., it is never freed (see (12.1)). This property correlates with GC where one may assume that a heap location r is not concurrently freed if it is just referenced by some running method. With hazard pointers one can make the same assumption if r is covered by a *validated* hazard pointer.

Before a process p validates a location r, however, it can be concurrently deallocated by another process q and arbitrarily reused even if p has already set its hazard pointer to r. This happens when the assignment $HPR_p := r$ is executed after the location has been retired by q, and q has already passed p's hazard pointer entry in its current traversal of HPR. Therefore, we omit any assertions about hazard pointers which are not validated yet. This differs from [80] who include such locations in their main correctness argument.

One difference between hazard pointers and GC is that while locations that are reachable from a root location can be concurrently freed if they are no longer covered by a validated hazard pointer, they would typically not be freed under GC, as long as their root is still used.

The next central property of hazard pointers ensures that retired locations are in the application's heap, but not in the lock-free data structure.

$$RL \subseteq (H - LDS) \tag{12.3}$$

This has two major consequences. First, deallocation steps are safe as they do not affect locations which are not in the application's heap. Second, succeeding validations (a location is in LDS at that time) imply that the validated location is currently not retired, hence not a deallocation candidate of any current scan.

Two further central properties of hazard pointers ensure that no ABA-problem occurs.

$$\textbf{if } r \in HPR_{valid} \textbf{ then } r \notin NEW \tag{12.4}$$

$$\textbf{if } \text{under GC: } H'(r) = H''(r) \textbf{ then if } r \in HPR_{valid}\text{: } \; H'(r) = H''(r) \tag{12.5}$$

(12.4) states that if a location r is covered by a validated hazard pointer, then it is not reused, i.e., it is not reinserted in the data structure which averts the ABA-problem. This property is also related to GC, where a heap location is not reused as long as it is referenced in some method. Hence, the environment assumption (12.5) holds: If the content of a heap location r is not concurrently changed in an environment with GC, then it is also unchanged when r is covered by a validated hazard pointer.

12.2.5. Instantiating the RG Parameters

First, we instantiate the RG parameters of proof obligation (6.2) where `PROC` is implemented according to our refinement-based proof method for linearizability. Thus we establish rely and invariant conditions that we then use in the refinement/lock-freedom proofs. The parameter procedure $\texttt{COP}(I, In; LSp, \mathcal{S}, Out)$ is one of push, pop, scan or

reset that is executed by a legal process (having an identifier $LSp.id \leq \text{MAXID}$) while illegal processes just skip. The global state $\mathcal{S}$ consists of the shared variables *Top*, *H*, *HPR* for the top-of-stack pointer, the application's heap and the hazard pointer record, whereas the local state *LSp* of a process is the tuple of all local variables *Id*, *New*, *SuccPush*, $Hazard_{pc}$, *LTop*, *OSucc*, *Scan*, $BefInc_{pc}$, *Lid*, *Lhp*, *PL*, *RL*. Similarly, the local state *LSq* of the other process *Idq* is defined.

The instance of the remaining predicates for idle states, invariants, relies and guarantees are derived from the generic properties (12.2) – (12.5). Properties in bold script are the corresponding verification conditions that are required under GC and which can be simply reused under hazard pointers.

Absence of Access Errors. The stack-specific counterpart of generic property (12.2) ensures that the snapshot pointer is allocated and covered by a validated hazard pointer in the hazardous code region of pop.

$$Hazard_{pc} \wedge \boldsymbol{LTop \neq null \rightarrow LTop \in H} \wedge HPR(Id) = LTop \tag{12.6}$$

The stack-specific version of (12.3) implies that retired locations are allocated and disjoint from the stack, where a standard reachability predicate checks whether a location r is in the stack.

$$\forall\, r \in RL.\ r \neq null \wedge r \in H \wedge \neg\, reach(Top, r, H) \tag{12.7}$$

(12.6) and (12.7) ensure that heap access errors do not occur in pop and scan.

To sustain (12.6) at all times in every possible execution, the validated hazard pointer $LTop = HPR(Id)$ used in a pop method of process p ($Hazard_{pc}$ holds, *Id* is the process identifier of p) must not be freed by any process q. The worst case is that q has retired *LTop*, just traverses *HPR*, but has not yet collected it ($LTop \in RLq - PLq$). Then q must not have passed the entry of p yet ($Lidq \leq Id$) and if it has reached p's entry, it must store *LTop* in the local variable *Lhpq* to ensure that it is collected. Invariant *ishazard* encodes this central criterion precisely:

$$\begin{array}{l} ishazard(LSp, LSq) \equiv \\ Hazard_{pc} \wedge LTop \in (RLq - PLq) \wedge Scanq \ \rightarrow \\ \quad \textbf{if } BefIncq_{pc} \textbf{ then } Lidq < Id \vee (Lidq = Id \wedge Lhpq = LTop) \textbf{ else } Lidq \leq Id \end{array}$$

Note that predicate $ishazard(LSp, LSq)$ is independent from the underlying data structure, here except for the critical snapshot reference *LTop* that is part of the local state. It can be directly used to ensure memory-safety for other lock-free data structures as well, as we illustrate below for a well-known queue algorithm.

Further simple properties that we use in the verification are the following: To sustain invariant (12.7) at all times, we must establish that retired lists are always duplicate free and pairwise disjoint. Otherwise, a retired list might contain a freed location after a deallocation step. Furthermore, three basic heap-disjointness properties are necessary: Removed locations, which are subsequently retired, must be disjoint from the stack and they must not be concurrently retired, plus, concurrently removed locations must be disjoint. To ensure that heap access faults do not occur in push either, we claim that

new nodes that have not been inserted yet, are always allocated and never concurrently retired, hence never freed.

Absence of Memory Leaks. The hazard pointers method avoids memory leaks, i.e., all heap locations are either in the lock-free data structure or referenced by a local pointer of some process. In terms of the stack, this global property is defined here as

$$\mathit{noleak}(\mathit{LSf}, \mathcal{S}) \equiv \forall\ r \in H.\ \mathit{reach}(\mathit{Top}, r, H) \vee \exists\ p.\ \mathit{refs}(r, \mathit{LSf}(p))$$

where function *LSf* encodes all local states and a process references local pointers according to its new, removed and retired locations.

$$\begin{aligned}&\mathit{refs}(r, \mathit{LSp}) \equiv\\ &\quad (\neg\ \mathit{SuccPush} \wedge \mathit{New} = r) \vee (\mathit{Hazard}_{pc} \wedge \mathit{OSucc} \wedge \mathit{LTop} = r) \vee r \in \mathit{RL}\end{aligned}$$

To avoid global reasoning, we decompose the absence of leaks to a process-local guarantee condition *noleaks*, which guarantees that each process step preserves absence of leaks for a reference r. (With this property, the guarantee becomes stronger then just the rely of other processes.)

$$\begin{aligned}&\mathit{noleaks}(\mathit{LSp}, \mathcal{S}, \mathit{LSp}', \mathcal{S}') \equiv \forall\ r.\\ &\qquad r \notin H \vee \mathit{reach}(\mathit{Top}, r, H) \vee \mathit{refs}(r, \mathit{LSp})\\ &\quad \rightarrow r \notin H' \vee \mathit{reach}(\mathit{Top}', r, H') \vee \mathit{refs}(r, \mathit{LSp}')\end{aligned}$$

ABA-prevention. The stack-specific version of (12.4) ensures that the validated snapshot in pop is not reused, thus it is disjoint from other new nodes.

$$\mathit{Hazard}_{pc} \wedge \neg\ \boldsymbol{\mathit{USuccq}} \rightarrow \boldsymbol{\mathit{LTop} \neq \mathit{Newq}} \tag{12.8}$$

The specialization of (12.5) yields the following rely condition which ensures that the snapshot's content is immutable in the hazardous code region of pop, to avoid an ABA-problem (as explained in the previous section).

$$\mathit{Hazard}_{pc}' \wedge \boldsymbol{\mathit{LTop}' \neq \mathit{null} \rightarrow H'(\mathit{LTop}') = H''(\mathit{LTop}')} \tag{12.9}$$

An ABA-problem does not happen in push as well, since the content of a new node remains unchanged.

$$\neg\ \boldsymbol{\mathit{SuccPush}' \rightarrow H'(\mathit{New}') = H''(\mathit{New}')} \tag{12.10}$$

To maintain rely (12.10) for the other process when the current push process updates the new node's next reference in line U8, new nodes must be disjoint.

$$\neg\ \boldsymbol{\mathit{SuccPush}} \wedge \neg\ \boldsymbol{\mathit{USuccq} \rightarrow \mathit{New} \neq \mathit{Newq}} \tag{12.11}$$

Verification conditions (12.6) and (12.7) are a main part of the invariant predicate *Inv*. Conditions *ishazard*, (12.8) and (12.11) are part of the symmetric disjointness predicate D, which is defined as:

$$D(\mathit{LSp}, \mathit{LSq}) \equiv \mathit{ishazard}(\mathit{LSp}, \mathit{LSq}) \wedge \mathit{ishazard}(\mathit{LSq}, \mathit{LSp}) \wedge (12.8) \wedge \ \ldots$$

Rely conditions (12.9) and (12.10) are the major part of R; guarantee G is defined to maintain R for the other process and the simple step-invariant *noleaks* which ensures that stack methods do not create memory leaks. Finally, the *Idle* predicate encodes the following local restrictions:

$$SuccPush \wedge OSucc \wedge \neg\, Hazard_{pc} \wedge \neg\, Scan \wedge \neg\, BefInc_{pc} \wedge Lid = 0 \qquad (12.12)$$

12.2.6. The Correctness Proofs

Sustainment of the Verification Conditions. The main effort of the case study is to prove the RG assertion for partial correctness according to (6.2) – sustainment of the verification conditions during steps of each individual method. Using the rules from Section 4.3, the proof resembles a Hoare-style proof of a sequential program. We use safety induction (4.3) to deal with CAS-loops. In the following, only the major arguments are sketched by case analysis over $Op \in \{\texttt{SCAN}, \texttt{POP}, \texttt{PUSH}, \texttt{RESET}\}$. Full proof details are available at [59].

$Op \equiv \texttt{SCAN}$: It is subtle to establish $ishazard(LSq, LSp)$ when the current process (with local state LSp) switches to the next hazard pointer entry during its traversal. This step must not miss a validated hazard pointer $OTopq$ of the other process q if the current process p has retired, but not yet collected it ($OTopq \in RL - PL$). If the snapshot Lhp of the current HPR entry is not null, we know from previous symbolic execution that it is in PL. If the current iteration examines q, *ishazard* before this step implies $Lhp = OTopq$, i.e., the validated hazard pointer has just been collected in the current iteration ($OTopq \in PL$), implying $ishazard(LSq, LSp)$.

In the deallocation step, *ishazard* ensures that the validated snapshot location of the other process is not freed (12.6). The proof is by contradiction: If the other process is in its hazardous code region and its snapshot pointer is in $RL - PL$, then *ishazard* before this step implies that the current process must not have finished its traversal. However, the current process is in its second scan loop already (technically, the contradiction is $\text{MAXID} + 1 = Lid \leq Idq \leq \text{MAXID}$).

$Op \equiv \texttt{POP}$: In the succeeding hazard pointer validation step, *ishazard* and (12.6) can be established, since the hazard pointer is in the data structure, hence allocated and not concurrently retired. Immediately after removal of the snapshot $LTop$ from the stack, we know from (12.7) that it can not be in the current process' retired list RL. Hence, we can establish (12.7) again in the step that retires a location, since both $LTop$ and RL are local.

$Op \equiv \texttt{PUSH}$: The allocation step resets the content of a new node. However, it does not affect allocated locations and thus neither rely condition (12.9) nor (12.10) of the other process are violated. We additionally establish $New \notin RL$ in this step which allows to prove disjointness of retired locations from the data structure (12.7), when the new node is added to the stack by CAS.

$Op \equiv \texttt{RESET}$: The reset of a hazard pointer entry is safe, since it happens outside of the hazardous code region in pop.

Preservation of Linearizability. The proof of linearizability (10.6) distinguishes between the four possible concrete methods. In case of the hazard pointer methods

scan and reset, each concrete step basically refines an abstract skip step. In particular, the deallocation step does not affect the stack, as retired locations are disjoint from the stack, according to (12.7).

The extended pop method still has one linearization point when it takes a snapshot of a null pointer, and otherwise if its CAS succeeds. Rely (12.9) ensures that the next reference of the snapshot node and its value are immutable. Thus, the successful CAS corresponds to an abstract pop and returns the correct value. In case of a push method, the linearization point is the successful CAS. Rely (12.10) ensures that the initial value of the new node and its next reference are immutable. Hence, the successful CAS corresponds to an abstract push of the invoked value.

Preservation of Lock-Freedom. According to proof obligations (11.4) and (11.5), the proof of lock-freedom requires termination proofs for each data structure method if environment behavior is restricted according to U and if a step violates U a finite number of times (here ≤ 1). We instantiate the unchanged relation as identity of the top-of-stack pointer. It is then simple to show that push and pop terminate using similar arguments as in the previous section.

Since the scan method is wait-free, we can prove its termination without U. (In general, wait-free operations can be verified using trivial termination conditions $U \equiv \mathit{true}$.)

$$\texttt{SCAN}(; \mathit{LSp}, \mathcal{S}),\ \Box\ \mathit{LSp}' = \mathit{LSp}'' \vdash \Diamond\ \mathbf{last}$$

Termination of the first scan loop uses well-founded induction (2.3) over the term $\text{MAXID} - \mathit{Lid}$ which decreases in every iteration. Similarly, termination of the second loop follows by induction over the number of retired locations.

We have also verified the stack with hazard pointers using the following ownership annotations (similar to Section 12.1): New nodes are owned by a push process p and then get owner *stack* with their insertion to the stack. Similarly, when a reference is removed from the stack, it gets a pop process p as owner. Thus we could avoid to explicitly state disjointness properties, except for our central correctness property $\mathit{ishazard}(\mathit{LSp}, \mathit{LSq})$ which can not be expressed in terms of these ownership annotations only. Hence, completely avoiding the local state of another process from the specification is not easily possible in this case, but the use of ownership still leads to notable simplifications.

12.2.7. Non-Atomic Read/Write of Hazard Pointers

In this section, we consider a non-trivial variation of the hazard pointers technique which now reads and writes hazard pointer entries *non-atomically*. We outline the main adaptations of our previous specifications and sketch the essential differences in the proofs. Full proof details are available at [59].

Basic assignments are atomic in RGITL. To model non-atomic instructions, we use undeclared procedures that we specify with a temporal contract that describes their possible behaviors and replace the respective procedure calls with their contract formula in proofs. This is by exploiting the compositionality of the sequential composition

and iteration operators (1.1). Then the contract formula can be symbolically executed according to Theorem 2.2/Appendix A.1.

For instance, a non-atomic read method NA−READ that models a non-atomic assignment $LV :=_{\mathrm{NA}} SV$ can be specified with the following contract (given as an axiom)

$$\begin{aligned} & \texttt{NA}-\texttt{READ}(LV, SV) \\ \vdash \quad & \circ \Diamond\ \mathbf{last} \wedge \Box\ (\neg\ \mathbf{blocked} \wedge SV = SV') \\ & \wedge\ (\Box\ (\circ \neg\ \mathbf{last} \rightarrow SV' = SV'') \rightarrow \Box\ (\circ\ \mathbf{last} \rightarrow LV' = SV')) \end{aligned}$$

which terminates after an arbitrary (positive) finite number of non-blocking steps that leave the shared variable SV unchanged. If the environment never changes SV while the method executes ($\Box\ \ (\circ \neg\ \mathbf{last}\ \ \rightarrow SV' = SV'')$), then the method returns the value of SV with its last program transition $\Box\ (\circ\ \mathbf{last} \rightarrow LV' = SV')$. Otherwise, the local variable LV holds an arbitrary value when the method terminates. In general, after the last program transition of the method, the subsequent (last) environment transition possibly changes SV and we do not know how the current value of LV relates to SV when the method reaches its last state (**last**). Hence, atomic assignments as well as sequentially reading each single bit of SV are possible implementations of the specification above.

Similarly, we specify a non-atomic write operation $SV :=_{\mathrm{NA}} LV$ as

$$\begin{aligned} & \texttt{NA}-\texttt{WRITE}(LV; SV) \\ \vdash \quad & \circ \Diamond\ \mathbf{last} \wedge \Box\ (\neg\ \mathbf{blocked}) \\ & \wedge\ (\Box\ (\circ \neg\ \mathbf{last} \rightarrow SV' = SV'') \rightarrow \Box\ (\circ\ \mathbf{last} \rightarrow LV' = SV')) \end{aligned}$$

where the shared variable SV is only set to the input value LV if the environment leaves SV unchanged throughout the execution of the non-atomic write method.

To consider these non-atomic methods in the proofs, we adapt the specification from Figure 12.6 as follows: In method POP, we replace the atomic write assignment $HPR(Id) := LTop$ with the procedure call $\texttt{NA}-\texttt{WRITE}(LTop; HPR(Id))$. Similarly, we replace the atomic read assignment $Lhp := HPR(Lid)$ of SCAN with $\texttt{NA}-\texttt{READ}(Lhp, HPR(Lid))$.

Now to the RG specifications/proofs using these non-atomic read/write methods for the extended stack: Note that for hazard pointer entries which are single-writer shared variables, the non-atomic write method never has to cope with the case of interfering writes, i.e., $HPR(Id)$ is never changed concurrently while the current process Id is writing his entry in POP. This leaves the non-atomic read operation in SCAN as the only difficult case where the local snapshot variable Lhp that locally stores the hazard pointer entry $HPR(Lid)$ of some other process Lid, can be corrupted due to a concurrent write (of process Lid). To deal with this case, we add three simple

properties to our initial relies above:

$$
\begin{aligned}
&(\quad Scan' \wedge Hazardq_{pc}{}' \wedge OTopq' \in RL' \\
&\quad \rightarrow OTopq' = OTopq'' \wedge Hazardq_{pc}{}'' \vee OTopq' \neq OTopq'' \wedge OTopq' \notin RL') \\
&\wedge (Scan' \wedge OTopq' \notin RL' \rightarrow OTopq'' \notin RL') \\
&\wedge (Scan' \wedge \neg\, Hazardq_{pc}{}' \wedge Hazardq_{pc}{}'' \rightarrow OTopq'' \notin RL')
\end{aligned}
$$

According to the first conjunct, when the current process performs a scan and has retired the local snapshot of the other process ($OTopq' \in RL' = RL''$) which is in its critical code region $Hazardq_{pc}{}'$, the other process either i) leaves its snapshot unchanged and stays in his critical region, or ii) it changes its snapshot which is then no longer in the retired list of the current process. The second/third conjunct then only ensure that if the other process takes a new snapshot *OTopq* and possibly reenters its critical code region, then his new snapshot is not in the retired list of the current process.[2]

Finally, we briefly discuss how these additional rely properties make it possible to propagate the central condition *ishazard* in the RG proof of the `SCAN` method: When the current process reads the hazard pointer entry of some other process during a scan, two cases are possible in an environment transition according to i)/ii) above: In case i), the other process stays in its critical region and thus its hazard pointer entry also remains unchanged. (The invariant ensures that $OTopq = HPR(Lid)$ in critical regions.) The proof then largely corresponds to the one with atomic operations, but one additional step of symbolic execution is necessary to discern whether the non-atomic method terminates now, or in a future state. In the critical case ii), the other process refreshes its snapshot by atomically reading the top-of-stack pointer. This new snapshot is not in the retired list *RL* of the current process, since it is disjoint from the stack data structure. Then the additional relies ensure that none of the following environment transitions can put the new snapshot into *RL*. Thus, the hazard pointer that is set by an interfering write operation will not be deallocated by the current process.

12.2.8. The Michael-Scott Queue with Hazard Pointers

Our central correctness condition for the hazard pointers method (predicate *ishazard*) can be directly used to also verify other algorithms with hazard pointers. This section sketches the essential adaptations of our specifications/proofs to verify the well-known lock-free MS queue that is extended with hazard pointers [68]. For full details refer to [59].

Figure 12.7 gives the extended queue algorithm. The queue is represented in the heap as a singly linked list of nodes (with element and next reference as for the stack) that are reachable from a shared dummy node *Head*: *ref* that has an arbitrary element.

[2] Note that to be able to state these additional properties in the rely predicate, it must have additional parameters for the local state of the other process before/after an environment transition. This leads to a slightly different state-local rule, but such minor differences are not essential here. For full details see [59].

```
ENQ(In; LSp, S) {
  choose New_0 with (New_0 ≠ null ∧ New_0 ∉ dom(H)) in {
    New := New_0, dom(H) := dom(H) ∪ {New_0}, H(New_0) := (In, null), ESucc := false;
    while ¬ ESucc do {
      ETail := Tail, HazardETail_pc := false;
      HPR(Id)(0) := ETail;
      if* ETail = Tail {   /* Validation */
        HazardETail_pc := true;
        ENxt := H(ETail).nxt;
        if ENxt ≠ null {
          if* ETail = Tail {Tail := ENxt}   /* CAS Shift Lagging Tail */
        } else {
          if* H(ETail).nxt = null {H(ETail).nxt := New, ESucc := true}   /* CAS ENQ */
      }};
    if* ETail = Tail {Tail := New};   /* CAS Shift Lagging Tail */
    HazardETail_pc := false}}

DEQ(In; LSp, S, Out) {
  let LOut = empty, DSucc = false in {
    while ¬ DSucc do {
      DHead := Head, HazardDHead_pc := false, HazardDNxt_pc := false;
      HPR(Id)(0) := DHead;
      if* DHead = Head {   /* Validation */
        HazardDHead_pc := true;
        DTail := Tail;
        DNxt := H(DHead).nxt;
        HPR(Id)(1) := DNxt;
        if* DHead = Head {   /* Validation */
          HazardDNxt_pc := true;
          if DNxt = null {
            DSucc := true, HazardDHead_pc := false, HazardDNxt_pc := false
          } else {
            if DHead = DTail {
              if* DTail = Tail {Tail := DNxt}   /* CAS Shift Lagging Tail */
            } else {
              if* DHead = Head {Head := DNxt, DSucc := true}   /* CAS DEQ */
            }}}};
      if DNxt ≠ null {
        LOut := H(DNxt).el;
        RL := RL + DHead, HazardDHead_pc := false, HazardDNxt_pc := false;
      };
      Out := LOut}}
```

Figure 12.7.: Specification of MS Queue with Hazard Pointers.

The end of the list is represented by a shared tail pointer $Tail\colon ref$ which can lag at most one node behind the last node of the list. Whenever an enqueue method `ENQ` encounters a lagging tail pointer, it first tries to shift it before it enqueues a new node at the end of the list. Method `DEQ` removes the dummy node from the list if it is not empty by shifting *Head* to its successor node if the queue is not empty. Otherwise, it returns *empty* and if the method finds a lagging tail pointer in a queue with just one element, then it tries to shift it. (For a more detailed explanation of the basic algorithms under GC see, e.g., [41] or [7, 26, 93, 94] for a slightly improved version.)

Method `ENQ` requires one hazard pointer for its snapshot variable *ETail* to avoid memory errors/ABA problems. For the dequeue method, both the snapshot of the head-of-queue pointer *DHead* and its next reference *DNxt* must be covered by a hazard pointer. Since the critical code regions where these two local pointers are used overlap, two hazard pointers per process are necessary. Therefore, we now model the hazard pointer record as a function $HPR\colon nat \to (nat \to ref)$ and we use three boolean flags $HazardETail_{pc}$, $HazardDHead_{pc}$ and $HazardDNxt_{pc}$ to mark the critical code regions of the local snapshot variables *ETail*, *DHead* and *DNxt*, respectively.

Furthermore, we generalize the scan method to the case of 2 hazard pointers per process by replacing its first loop from Figure 12.6 with the following nested loop

$$
\begin{array}{l}
\textbf{while } Lid \leq \text{MAXID } \textbf{do } \{ \\
\quad \textbf{let } Ix = 0 \ \textbf{in } \{ \\
\quad\quad \textbf{while } Ix < 2 \ \textbf{do } \{ \\
\quad\quad\quad Lhp := HPR(Lid)(Ix),\ BefInc_{pc}(Ix) := true; \\
\quad\quad\quad \textbf{if } Lhp \neq null \textbf{ then } \{ \\
\quad\quad\quad\quad PL := Lhp + PL \\
\quad\quad\quad \}; \\
\quad\quad\quad Ix := Ix + 1 \\
\quad\quad \}\}; \\
\quad Lid := Lid + 1,\ BefInc_{pc} := \lambda\, b.\ false\};
\end{array}
$$

where the auxiliary flag $BefInc_{pc}$ is now a function from naturals to booleans.

For the verification of the extended queue algorithm, predicate $ishazard(LSp, LSq)$ does not have to be changed, but we must strengthen our definition of the disjointness predicate $D(LSp, LSq)$ for the queue to consider this predicate for each snapshot/auxiliary flag

$$
\begin{array}{rl}
& ishazard(HazardETail_{pc}, ETail, BefIncq_{pc}(0), \dots) \\
\wedge\ & ishazard(HazardDHead_{pc}, DHead, BefIncq_{pc}(0), \dots) \\
\wedge\ & ishazard(HazardDNxt_{pc}, DNxt, BefIncq_{pc}(1), \dots)
\end{array}
$$

plus symmetric versions. The remaining queue-specific RG conditions are simply derived from their version under garbage collection (see [59]), as we have explained above for the stack. The central correctness arguments for the extended queue are then similar to the arguments that we have used before.

12.3. Summary

In this chapter we have applied our verification approach to mechanically verify linearizability and lock-freedom for lock-free data structures that are extended with modification counters and hazard pointers, respectively. To our knowledge, no (successful) mechanized verification of these algorithms has been described in the literature before. For an extended discussion on related work see Chapter 13. We have illustrated how ownership annotations (and separation logic) can simplify the specifications/proofs in these cases. Furthermore, we have shown that state-local reasoning with two explicit local states can be sufficient for algorithms with hazard pointers. For the first time, we have verified a data-structure correct with hazard pointers that are read (and written) non-atomically, and we have illustrated how to adapt our proof method to other algorithms with hazard pointers using a non-trivial queue example.

The next chapter discusses related verification approaches and concludes with possible future work.

13. Related Work and Conclusion

Comparison with Own Previous Work

The Ph.D. thesis [6] takes a first step to define a refinement-based proof method for linearizability in RGITL, similar to our proof method in Section 10.1. Moreover, it defines a proof method for lock-freedom. This work improves/extends this basic approach in various ways:

- We now introduce a generic proof method for linearizability according to Section 9.3 and derive its correctness with respect to the original definition of linearizability [48].
- From this generic proof method, we derive our refinement-based method in Section 10.1. In contrast, the proof method in [6] makes no formal connection to linearizability and argues informally only that an algorithm-specific refinement is sufficient.
- Proofs of linearizability in [6] are restricted to the verification of algorithms with internal linearization points, while we can now also verify complex external linearization points using our generic proof method (see Section 9.4).
- The definition of lock-freedom in [6] is based on weak-fair interleaving and does not consider the case where a process is never executed again. Instead, we now consider unfair interleaving [94], which leads to a different soundness proof that requires different fairness rules (see Section 11.3).
- The proof methods in [6] are state-global and do not consider state-local proofs of linearizability/lock-freedom (based on our state-local RG rule 6.2), nor do they apply techniques such as ownership or separation logic to simplify the specifications/proofs.
- The proof methods in [6] are only applied to a standard stack/queue algorithm under the assumption of garbage collection, based on [93]. Now we provide state-local linearizability/lock-freedom proofs for extended versions of these algorithms that use challenging lock-free memory reclamation techniques (see Sections 12.1 and 12.2). Moreover, we have verified further challenging algorithms correct such as the wait-free/lock-based multiset implementations or the lock-free set (Sections 9.4, 10.2 and 11.4).

In other work [21–24, 88], we have developed a rather different approach for proving linearizability in KIV, using predicate logic (PL) instead of temporal logic (TL) and Owicki/Gries-style decomposition [78] rather than RG reasoning. The PL approach is not in the scope of this thesis, but our proof methods for linearizability that we develop here, ground on this work. In particular, the PL approach defines a complete proof method for linearizability [88] that we integrate with RG reasoning in Section 9.3. Moreover, it gives a local proof method for linearizability [24] that uses a status function to encode linearization points. We compare the PL/TL approaches in more detail in the following:

- The current PL approach manually encodes programs as transition systems with program counters. This manual encoding is error-prone. In RGITL we use an abstract programming language to encode programs and thus the actual algorithm and the verification conditions are typically much easier to read/correct. Occasionally, however, auxiliary variables must be added to characterize code ranges.
- The PL approach uses Owicki/Gries-style reasoning and requires to explicitly encode *local* state information using program counters in the invariants. The TL approach can largely avoid such low-level encodings using symbolic execution and RG reasoning where local state information is automatically computed and propagated.
- Similar to the PL approach [88], the core arguments why an algorithm is linearizable using our generic TL proof method (Section 9.3) boil down to the verification of predicate logic lemmas that show a backward simulation for individual program transitions (see Section 9.4).
- Our refinement-based proof method (Section 10.1) can not handle external linearization, while the local proof method from [24] can if a linearization does not change the abstract state. In contrast, an encoding of linearization points using program counters and a status function is not necessary in our refinement-based method, since the linearization status is implicitly given by the abstract rest program during symbolic execution. (Deriving state-local proof obligations in RGITL for algorithms with external linearization points that leave the state unchanged, similar to [24], is left for possible future work.)
- The PL approach (like many other approaches) does not consider liveness verification.

Comparison with Other Approaches

Initial work on mechanically verifying linearizability with PVS is described in [19, 26] and their work also found bugs in lock-free data structures [18, 25]. Their approach applies a reduced version of IO-automata [65] to verify linearizability using an encoding

of programs as transition systems and forward/backward simulations as basic proof techniques. Their proofs are over the entire system state. Our proofs are compositional and typically state-local. In particular, we can avoid the use of backward simulation for internal potential linearization points using our refinement-based proof method. In [42] a pen-and-paper proof of linearizability is given for a stack with modification counters. The proof is based on trace reduction and incremental refinement. Their algorithm reuses memory from an abstract set of free locations. Instead, we consider an implementation of a free stack and give mechanized proofs of memory-safety, ABA prevention, linearizability and lock-freedom (Section 12.1).

The work of [19, 26] also mentions the use of the model-checker SPIN [52] to find bugs in linearizable algorithms. A more detailed exposition of model-checking linearizability is given in [108]: While model-checking and other "light-weight verification" approaches (such as testing) can in general not provide full proofs that consider all possible executions, they can find some bugs relatively fast. Thus they are useful auxiliary techniques to check the soundness of a concurrent implementation. Other approaches for checking linearizability are for example [11, 107].

[29] proposes an interesting interactive proof method for linearizability based on their calculus of atomic actions [30]. Their approach is mechanized in the QED tool and iteratively performs abstraction/reduction steps starting with an implementation until the resulting program is (ideally) equal to the abstract specification. Linearization points do not have to be specified. Instead, the proof engineer must identify code blocks that can be safely merged to get a bigger atomic code block. To ensure that the merge is safe requires to verify properties about the commutativity of individual instruction pairs. Such proof obligations are also typical for our proofs of linearizability using the generic proof method, see Section 9.4. Their approach tries to automatically discard these commutativity proofs using an SMT-solver, while we discard them interactively. While the multiset with fine-grained locking (but without a delete operation) is one of their examples that has been solved in QED using ownership, a delete operation has not been considered. [30] also mentions an attempt to prove a variant of the stack with hazard pointers, but unfortunately no details are given why it has not been successful.

Lots of work has focussed on verifying concurrent algorithms using separation logic, e.g., [33, 37, 77, 80, 106]. In particular, RGSep [106] is a program logic that combines RG reasoning and separation logic to achieve heap-modular reasoning about partial correctness of concurrent programs. Different from RGSep, heaps are not part of the semantics of RGITL and our abstract programming language makes no restrictions on the possible modifications of a program to a heap variable. This makes it more difficult for us to achieve full heap-modular reasoning without specializing the semantics and we must explicitly specify/verify which part of the heap a program leaves unmodified. We have found ownership annotations helpful to achieve better heap modularity, since they make it possible to easily ensure that a program leaves portions of the heap unchanged that have a distinct owner. Similarly, in RGSep local/shared assertions refer to either the local heap of one process or the shared heap and ownership can be transferred from the local state to the shared state and vice versa. The logic has been implemented in an automatic verifier for linearizability [104] that verifies several non-

trivial (typically list-based) algorithms linearizable. Different from RGITL, *deriving* compositional proof methods as well as refinement/liveness proofs are not in the scope of RGSep.

In [80], heap-modular reasoning for concurrent programs is achieved by a combination of concurrent separation logic and fractional permissions. They illustrate their approach by manually verifying partial-correctness of a variant of the stack with hazard pointers. Their heap-modular proofs use non-trivial history variables to capture the central correctness arguments of hazard pointers. The need for history variables is reduced in [37] where partial correctness of another version of the stack with hazard pointers is shown in a program logic with temporal past operators. Both approaches [37, 80] quantify over all processes in their specifications/proofs for the stack and thus do not fully exploit the symmetry of the underlying hazard pointers technique. Recent work [40] extends RGSep with temporal logic to reason about memory reclamation and the approach is manually applied to verify partial correctness of the stack with hazard pointers.

The recent proof obligations in [64] for linearizability cover a bigger class of algorithms than our refinement-based proof method, but specifications/proofs are no longer strictly local. Instead their manual proofs essentially record global information about the operations that have linearized (similar to our set R in Section 9.3) as auxiliary annotations in the code.

Recent work [47] gives a complete approach for proving linearizability for a specific type of purely blocking queue algorithms. Purely blocking essentially means that the global state may not change in infinite runs. (For instance, our multiset with fine-grained locking is not purely blocking, since infinite executions in `INSERTPAIR`, where a process eventually starves waiting for a lock, do not always leave the array unchanged.) Our proof methods are not data structure specific. It would be nevertheless interesting to invent similar reductions for other data structures, e.g., sets/multisets, and to investigate how their proofs (really) compare to our refinement based proofs.

An automated decomposition technique for lock-freedom is given in [38]. The main idea there is that lock-freedom can be reduced to the proof of termination of a reduced system $\texttt{SPAWN}_k^{red}$ that runs in isolation and where the infinite iterations of each process are removed, i.e., each process calls $\texttt{COP}_p$ just once. It is argued informally that this reduction is correct, while we derive the correctness of our decomposition on calculus level (see, e.g., Section 11.3). Moreover, [38] defines a calculus to reduce the termination of $\texttt{SPAWN}_k^{red}$ to local termination conditions for $\texttt{COP}_p$ based on RG rules with *temporal* RG conditions, whereas our conditions are binary predicates. Their action inference algorithm can be roughly seen as an automatic way to construct our predicate U. This is also the only work that mentions the lock-free set [70] as one of the algorithms that they can automatically verify. However, they do not provide enough information to understand how their tools can actually identify the underlying induction principle that is required for this proof.

[17] describe a technique for proving lock-freedom based on a low-level encoding of programs with program counters and an explicitly constructed well-founded order on these counters. Their proof method requires to show that each program transition

either leads to progress (which roughly corresponds to those steps that violate our relation U) or reduces the state w.r.t. the defined order. In our approach, this well-founded order is given implicitly when symbolically executing a (sequential) program. Studying the relationship between our approach with termination conditions U and the general idea of using well-founded orders would be interesting. A more recent manual approach for proving lock-freedom is [51] based on concurrent separation logic.

Conclusion

In Part III we have introduced compositional proof methods for linearizability/lock-freedom and we have illustrated how to apply them to verify various intricate concurrent data structure implementations correct. We have shown how the use of state-local reasoning and other techniques such as ownership annotations and separation logic can lead to simpler specifications/proofs. A difference between our work and other related approaches is that while others use calculi and tools for specific algorithms/properties (e.g., linearizability proofs for linked-list based algorithms), we consider both safety and liveness verification in the uniform setting of RGITL. Thus we can mechanize the correctness proofs for our "domain-specific" proof methods, while others argue informally that their reductions are correct or give pen-and-paper proofs only. We also use RGITL to verify specific programs correct: In particular, we have applied our proof methods (for linearizability/lock-freedom) to a number of challenging case studies, e.g., our highly concurrent multiset algorithm with intricate external linearization points that potentially linearize several other running processes (Section 9.4), or the lock-free stack/queue that use the intricate memory reclamation technique of hazard pointers [68] (Section 12.2).

There are several possible directions for future work: Mechanizing a completeness proof for our generic proof method from Section 9.3 is left for future work. Furthermore, achieving more modular specifications/proofs is still an open issue as explained above. For instance, our proof of the wait-free multiset (Section 9.4) considers the full program state with an arbitrary finite number of local states. We leave it for possible future work to determine whether a verification of the multiset with only a fixed small number of local states is also possible (by exploiting the symmetry of the underlying algorithms, e.g., lookup and delete operations that return false have identical behaviors). Another interesting question is how to integrate the ideas of atomic code blocks [30] into our proof methods, since this could result in simpler invariants and proofs.

Our current proof methods are based on sequentially consistent memory, so adapting these methods to weak memory models [16] would be another possible direction. Investigating on compositional proof methods for blocking progress properties such as deadlock-freedom [49] would be another option. Moreover, applying our proof methods to algorithms for software transactional memory (STM) [49] would be another possibility, e.g., it seems as if the correctness of some STM algorithms can be roughly verified based on linearizability where sequences of operations that are in a transaction either fail or execute successfully in one atomic step. Finding proof methods for further cor-

rectness conditions is another possible direction, as well as applying our concurrency theory to different application domains such as flash file systems [32].

Bibliography

[1] Abadi, M., Lamport, L.: Conjoining specifications. ACM TOPLAS, pp. 507–534 (1995)

[2] Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: Proc. of Euro-Par (2). pp. 151–162 (2010)

[3] Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Proc. of CAV. pp. 477–490 (2007)

[4] Apt, K.R., de Boer, F., Olderog, E.R.: Verification of Sequential and Concurrent Programs. Springer LNCS (2009)

[5] Balser, M.: Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction. Ph.D. Thesis, University of Augsburg (2005)

[6] Bäumler, S.: Modulares Beweisen temporallogischer Eigenschaften paralleler Programme. Ph.D. Thesis, University of Augsburg (2010)

[7] Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. FAC Journal 23(1), 91–112 (2011)

[8] Börger, E., Stärk, R.F.: Abstract State Machines — A Method for High-Level System Design and Analysis. Springer LNCS (2003)

[9] Bormer, T., Brockschmidt, M., Distefano, D., Ernst, G., Filliâtre, J.C., Grigore, R., Huisman, M., Klebanov, V., Marché, C., Monahan, R., Mostowski, W., Polikarpova, N., Scheben, C., Schellhorn, G., Tofan, B., Tschannen, J., Ulbrich, M.: The COST IC0701 Verification Competition 2011. In: Proc. of FoVeOOS. pp. 3–21. Springer LNCS (2011)

[10] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. of LICS. pp. 428–439. IEEE Computer Society Press (1990)

[11] Burckhardt, S., C.Dern, Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proc. of PLDI. pp. 330–340. ACM (2010)

[12] Burstall, R.M.: Program proving as hand simulation with a little induction. Information Processing 74 pp. 309–312 (1974)

[13] Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., R.Alur: Model checking of linearizability of concurrent list implementations. In: Proc. of CAV. vol. 4144, pp. 465–479, Springer LNCS (2010)

[14] Click, C.: Towards a scalable non-blocking coding style. Talk at JavaOne conference. Available at `http://www.azulsystems.com/events/javaone_2008/` (2008), Code available at `http://high-scale-lib.git.sourceforge.net/`

[15] Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Proc. of CAV. pp. 480–494. Springer LNCS (2010)

[16] Cohen, E., Schirmer, B.: From total store order to sequential consistency: A practical reduction theorem. In: Proc. of ITP. pp. 403–418 (2010)

[17] Colvin, R., Dongol, B.: A general technique for proving lock-freedom. Science of Computer Programming 74(3), 143–165, Elsevier (2009)

[18] Colvin, R., Groves, L.: Formal verification of an array-based non-blocking queue. In: Proc. of ICECCS '05. pp. 507–516. IEEE Computer Society, Washington, DC, USA (2005)

[19] Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: In Proc. of CAV. vol. 4144, pp. 475–488. Springer LNCS (2006)

[20] Derrick, J., Dongol, B., Schellhorn, G., Tofan, B., Travkin, O., Wehrheim, H.: Quiescent consistency: Defining and verifying relaxed linearizability. In: Proc. of FM. vol. 8442, pp. 200–214, Springer LNCS (2014)

[21] Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizabilty via non-atomic refinement. In: Proc of iFM, vol. 4591, pp. 195–214. Springer LNCS (2007)

[22] Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanising a correctness proof for a lock-free concurrent stack. In: Proc. of FMOODS, vol. 5051, pp. 78–95. Springer LNCS (2008)

[23] Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. In: TOPLAS. vol 33(4), pp. 1–43 (2011)

[24] Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisabilty with potential linearisation points. In: Proc. of FM. vol. 6664, pp. 323–337. Springer LNCS (2011)

[25] Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Jr., G.L.S.: DCAS is not a silver bullet for non-blocking algorithm design. In: Proc. of SPAA '04. pp. 216–224. ACM, New York, NY, USA (2004)

[26] Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: Proc. of FORTE 2004. vol. 3235, pp. 97–114, Springer LNCS (2004)

[27] Dongol, B., Derrick, J., Hayes, I.J.: Fractional permissions and non-deterministic evaluators in interval temporal logic. ECEASST 53 (2012)

[28] Dongol, B.: Formalising progress properties of non-blocking programs. In: Proc. of FMSE. vol. 4260, pp. 284–303. Springer LNCS (2006)

[29] Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: Proc. of TACAS. vol. 6015. pp. 296–311. Springer LNCS (2010)

[30] Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. SIGPLAN Not. 44, 2–15 (2009)

[31] Elmas, T., Tasiran, S., Qadeer, S.: Vyrd: Verifying concurrent programs by runtime refinement-violation detection. In: Proc. of PLDI. pp. 27–37. ACM, New York, USA (2005)

[32] Ernst, G., Schellhorn, G., Haneberg, D., Pfähler, J., Reif, W.: Verification of a Virtual Filesystem Switch. In: Proc. of VSTTE. vol. 8164, pp. 242–261. Springer LNCS (2014)

[33] Feng, X.: Local rely-guarantee reasoning. In: Proc. of POPL. pp. 315–327 (2009)

[34] Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. TCS. vol. 411(51-52), pp. 4379 – 4398 (2010)

[35] Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Proc. of POPL. pp. 256–267. ACM, New York, USA (2004)

[36] Floyd, R.W.: Assigning meanings to programs. In: Proc. of Symposium on Applied Mathematics. vol. 19, pp. 19–32. American Mathematical Society (1967)

[37] Fu, M., Y. Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: Proc. of CONCUR. pp. 388–402 (2010)

[38] Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: Proc. of POPL. pp. 16–28. ACM (2009)

[39] Gotsman, A., Yang, H.: Linearizability with ownership transfer. In: Proc. of CONCUR. pp. 256–271, Springer LNCS (2012)

[40] Gotsman, A., Rinetzky, N., Yang, H.: Verifying concurrent memory reclamation algorithms with grace. In: Proc. of ESOP. pp. 249–269, Springer, Berlin, Heidelberg (2013)

[41] Groves, L.: Verifying michael and scott's lock-free queue algorithm using trace reduction. In: Proc. of CATS. pp. 133–142. CATS, Australian Computer Society, Inc. (2008)

[42] Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. FAC Journal. vol. 21(1–2), pp. 187–223 (2009)

[43] Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)

[44] Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proc. of DISC. pp. 300–314, Springer, London (2001)

[45] Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. In: Proc. of OPODIS. vol. 3974, pp. 305–309. Springer LNCS (2005)

[46] Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proc. of SPAA '04. pp. 206–215. ACM Press, New York, NY, USA (2004)

[47] Henzinger, T., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: Proc. of CONCUR. pp. 242–256. Springer LNCS (2013)

[48] Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS. vol. 12(3), 463–492 (1990)

[49] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)

[50] Hoare, C.A.R.: An axiomatic basis for computer programming. vol. 12(10). pp. 576–580, Communications of the ACM (1969)

[51] Hoffmann, J., Marmar, M., Shao, Z.: Quantitative Reasoning for Proving Lock-Freedom. In: Proc. of LICS, pp. 124–133, IEEE (2013)

[52] Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison Wesley (2003)

[53] Jones, C.B.: Specification and design of (parallel) programs. In: Proc. of IFIP'83. pp. 321–332. North-Holland (1983)

[54] Jones, C.B.: Accommodating interference in the formal design of concurrent object-based programs. Formal Methods in System Design 8(2), 105–122 (1996)

[55] King, J.C.: A Program Verifier. Ph.D. thesis, Carnegie Mellon University (1970)

[56] Web presentation of linearizability theory and a lazy set (2011), `http://www.informatik.uni-augsburg.de/swt/projects/possibilities.html`

[57] Presentation of higher-order specifications of RGITL and soundness proofs. `http://www.informatik.uni-augsburg.de/swt/projects/RGITL.html` (2013)

[58] Presentation of KIV proofs for decomposition theory and lock-free stacks with modification counters (2013), `https://swt.informatik.uni-augsburg.de/swt/projects/avocs13.html`

[59] Presentation of proofs for some derived rules and case studies in RGITL (2013). `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`

[60] Web presentation of KIV proofs for linearizability theory and multiset with fine-grained locking (2013), `https://swt.informatik.uni-augsburg.de/swt/projects/MultiSet.html`

[61] Web presentation of KIV proofs for linearizability theory and wait-free multiset (2014), `https://swt.informatik.uni-augsburg.de/swt/projects/ifm14.html`

[62] KIV homepage: `http://www.informatik.uni-augsburg.de/swt/kiv`

[63] Lamport, L.: The temporal logic of actions. ACM TOPLAS, vol. 16(3), 872–923 (May 1994)

[64] Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: Proc. of PLDI. pp. 459–470. ACM (2013)

[65] Lynch, N., Vaandrager, F.: Forward and Backward Simulations – Part I: Untimed systems. Information and Computation 121(2), pp. 214–233 (1995)

[66] Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems – Safety. Springer (1995)

[67] Massalin, H., Pu, C.: A Lock-Free Multiprocessor OS Kernel. Tech. Rep. CUCS-005-91, Columbia University (1991)

[68] Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. Transactions on Parallel and Distributed Systems, vol. 15(6), pp. 491–504, IEEE (2004)

[69] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of PODC. pp. 267–275, ACM (1996)

[70] Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: In Proc. of SPAA '02. pp. 73–82. SPAA '02, ACM (2002)

[71] Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE, Trans. on Software Eng. 7, 417–426 (1981)

[72] Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: SPAA. pp. 253–262. ACM (2005)

[73] Moore, J.S.: A mechanically checked proof of a multiprocessor result via a uniprocessor view. Formal Methods in System Design. vol. 14. pp. 213–228 (1999)

[74] Moszkowski, B.: Executing Temporal Logic Programs. Cambr. Univ. Press (1986)

[75] Moszkowski, B.: Compositional reasoning about projected and infinite time. In: Proc. of ICECCS. pp. 238–245. IEEE Computer Society (1995)

[76] Müller, P.: Modular Specification and Verification of Object-Oriented Programs. vol. 2262. Springer LNCS (2002)

[77] O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)

[78] Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf. 6, 319–340 (1976)

[79] Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Commun. ACM 19 (1976)

[80] Parkinson, M., Bornat, R., O'Hearn, P.: Modular verification of a non-blocking stack. SIGPLAN Not. 42(1), 297–302 (2007)

[81] Pfaehler, J.: Formal Verification of Efficient Data Structures for Multi-Core Processors. Master Thesis, University of Augsburg (2012)

[82] Pnueli, A.: The temporal logic of programs. In: Proc. 18th Ann. IEEE Symp. on the Foundation of Computer Science (FOCS). pp. 46–57. IEEE Computer Society Press (1977)

[83] Prensa Nieto, L.: The rely-guarantee method in Isabelle /HOL. In: Proc. of ESOP'03. vol. 2618. pp. 348–362. Springer LNCS (2003)

[84] Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Proc. of Automated Deduction—A Basis for Applications, vol. II, pp. 13–39. Kluwer (1998)

[85] Reynolds, J.C.: Separation Logic: A logic for shared mutable data structures. In: Proc. of LICS. pp. 55–74. IEEE Computer Society, USA (2002)

[86] de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. No. 54 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (2001)

[87] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: TOPLAS. vol. 24, pp. 217–298. ACM (2002)

[88] Schellhorn, G., Derrick, J., Wehrheim, H.: How to prove algorithms linearizable. In: Proc. of CAV. vol. 7358, pp. 243–259. Springer LNCS (2012)

[89] Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: A temporal logic framework for compositional reasoning about interleaved programs. Annals of Mathematics and Artificial Intelligence (AMAI) Journal, vol. 71. pp. 131–174. Springer (2014)

[90] Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of TIME. pp. 99–106. IEEE Computer Society Press (2011)

[91] Shavit, N.: Data Structures in the Multicore Age. In: Commun. ACM. vol. 54, pp. 76–84, ACM (2011)

[92] Stølen, K.: A method for the development of totally correct shared-state parallel programs. In: Proc. of CONCUR. vol. 527, pp. 510–525. Springer LNCS (1991)

[93] Tofan, B.: Correctness Analysis Of Lock-Free Algorithms For Multi-Core Architectures. Master Thesis, University of Augsburg (2009)

[94] Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: Proc. of MPC 2010. vol. 6120. pp. 377–396. Springer LNCS (2010)

[95] Tofan, B., Schellhorn, G., Ernst, G., Pfähler, J., Reif, W.: Compositional verification of a lock-free stack with RGITL. In: et al., S.M. (ed.) In Proc. of AVoCS. vol. 66. ECEASST (2013)

[96] Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Proc. of ICTAC. vol. 6916. pp. 239–255. Springer LNCS (2011)

[97] Tofan, B., Schellhorn, G., Reif, W.: Local rely-guarantee conditions for linearizability and lock-freedom. Pre-Proc. of FoVeOOS, Karlsruhe Reports in Informatics 2011-26, KIT (2011)

[98] Tofan, B., Schellhorn, G., Schödel, S., Reif, W.: A Compositional Proof Method for Linearizability Applied to a Wait-Free Multiset. In: Proc. of iFM. vol. 8739. pp. 357–372. Springer LNCS (2014)

[99] Tofan, B., Travkin, O., Schellhorn, G., Wehrheim, H.: Two approaches for proving linearizability of multiset. Science of Computer Programming Journal, Elsevier to appear (2014)

[100] Tofan, B., Schellhorn, G., Reif, W.: Rely-Guarantee Reasoning in RGITL: Illustrating Compositional Proofs of Partial/Total Correctness and Absence of Deadlock. Acta Informatica, Springer, submitted (2014)

[101] Travkin, O., Wehrheim, H., Schellhorn, G.: Proving linearizability of multiset with local proof obligations. In: Lüttgen, G., Merz, S. (eds.) Proc. of AVoCS. vol. 53. ECEASST (2012)

[102] Treiber, R.K.: System programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center (1986)

[103] Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. Thesis, University of Cambridge (2007)

[104] Vafeiadis, V.: Automatically proving linearisability. In: Proc. of CAV. vol. 6174. pp. 450–464. Springer LNCS (2010)

[105] Vafeiadis, V.: RGSep action inference. In: Proc. of VMCAI 2010. vol. 5944. pp. 345–361. Springer LNCS (2010)

[106] Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: Proc. of CONCUR. vol. 4703, pp. 256–271. Springer LNCS (2007)

[107] Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Proc. of CAV. pp. 465–479. Springer LNCS (2010)

[108] Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proc. of SPIN Workshop on Model Checking Software. pp. 261–278. Springer LNCS (2009)

[109] Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. FAC Journal 9(2), 149–174 (1997)

A. Appendix

A.1. Executing Sequential Composition and Interleaving

This appendix describes the symbolic execution of the sequential composition and weak-fair/unfair interleaving operators for regular formulas φ, ψ. In particular, we briefly outline how to establish step form for these operators. Formulas τ and τ_i are step formulas in the following.

Sequential Composition

For a program $[\alpha; \beta]_{\underline{x}}$ where α and β are regular programs, we establish step form as follows: Assume that α is already in the following step form (variables v_i are required for the case of local variables introduced by **let**, etc.)

$$\alpha \;\equiv\; (\tau_0 \wedge \mathbf{last}) \vee \bigvee_{i=1}^{n} (\exists\, \underline{v}_i.\ \tau_i \wedge \circ\, \varphi_i)$$

Then the following two distribution rules for $\vee$ and $\exists$ over sequential composition

$$\begin{array}{l}(\varphi_1 \vee \varphi_2); \psi \leftrightarrow (\varphi_1; \psi) \vee (\varphi_2; \psi) \\ \quad (\exists\, v.\ \varphi); \psi \leftrightarrow \exists\, v_0.\ \varphi_v^{v_0}; \psi, \quad \text{where } v_0 \notin (\mathrm{free}(\varphi) \setminus \{v\}) \cup \mathrm{free}(\psi)\end{array}$$

rewrite $[\alpha; \beta]_{\underline{x}}$ to the following formula

$$[\alpha; \beta]_{\underline{x}} \equiv (\tau_0 \wedge \mathbf{last}); [\beta]_{\underline{x}} \vee \bigvee_{i=1}^{n} \exists\, \underline{v}_i.\ (\tau_i \wedge \circ\, \varphi_i); [\beta]_{\underline{x}}$$

For the first disjunct above where α terminates, we use the following rule to get step form:

$$(\tau \wedge \mathbf{last}); \psi \;\leftrightarrow\; \tau^{\underline{x},\underline{x}}_{\underline{x}',\underline{x}''} \wedge \psi, \text{ where } \underline{x} = \mathrm{free}(\tau)$$

For the remaining disjuncts above, the following rule establishes step form:

$$(\tau \wedge \circ\, \varphi); \psi \;\leftrightarrow\; \tau \wedge \circ\, (\varphi; \psi)$$

Note that the equivalence rules above can often also handle the more general case of executing $[\varphi; \psi]_{\underline{x}}$ where φ/ψ are regular formulas. This case typically occurs when α/β is rewritten with a contract φ/ψ.

Interleaving

Since symbolic execution can only consider finite prefixes of a given interval where fairness is irrelevant, both interleaving operators share the same rules. Therefore, we only consider weak-fair interleaving in the following for brevity.

First of all, symbolic execution of an interleaving $\varphi \parallel \psi$ discerns whether φ or ψ is executed first according to the following rule:

$$\varphi \parallel \psi \leftrightarrow (\varphi \parallel^{<} \psi) \vee (\varphi \parallel^{>} \psi)$$

The rule uses an auxiliary operator $\varphi \parallel^{<} \psi$ that gives precedence to the first component

$$\begin{aligned} I \models \varphi \parallel^{<} \psi \quad \text{iff there are } & I_1, I_2, sc : \\ & I_1 \models \varphi \text{ and } I_2 \models \psi \text{ and } (I, sc) \in I_1 \oplus I_2 \text{ and } sc \text{ fair} \\ & \text{and if } sc \neq () \text{ then } sc(0) = 1 \text{ else } |I_1| = 0 \end{aligned}$$

and operator $\varphi \parallel^{>} \psi$ is simply defined as

$$\varphi \parallel^{>} \psi \;\equiv\; \psi \parallel^{<} \varphi$$

In the following, we will focus on the case where the first component φ is executed. (The other case is symmetric.) By induction, we assume that φ is in the following step form

$$\varphi \;\equiv\; (\tau_0 \wedge \mathbf{last}) \vee \bigvee_{i=1}^{n} (\exists\, \underline{u}_i.\ \tau_i \wedge b_i \wedge \circ\, \varphi_i)\ .$$

where the blocking information b_i for τ_i is either **blocked** or $\neg$ **blocked**.

Next we exploit compositionality to rewrite φ with the right-hand side of the equivalence above in $\varphi \parallel^{<} \psi$. The resulting disjunctions and existential quantifiers can be pulled out of the interleaving with the following two rules:

$$\begin{aligned} (\varphi_1 \vee \varphi_2) \parallel^{<} \psi &\leftrightarrow (\varphi_1 \parallel^{<} \psi) \vee (\varphi_2 \parallel^{<} \psi) \\ (\exists\, v.\ \varphi) \parallel^{<} \psi &\leftrightarrow \exists\, v_0.\ (\varphi_v^{v_0} \parallel^{<} \psi) \text{ where } v_0 \notin (\mathrm{free}(\varphi) \setminus \{v\}) \cup \mathrm{free}(\psi) \end{aligned}$$

This gives a first disjunct $(\tau_0 \wedge \mathbf{last}) \parallel^{<} \psi$ that corresponds to Case 1 of Definition 1.4 and can be handled by the following rule:

$$(\tau \wedge \mathbf{last}) \parallel^{<} \psi \leftrightarrow \tau^{\underline{x},\underline{x}}_{\underline{x}',\underline{x}''} \wedge \psi \text{ where } \underline{x} = \mathrm{free}(\tau)$$

For the other disjuncts, depending on b_i, one of the following rules is applicable

$$\begin{aligned} (\tau \wedge \neg\, \mathbf{blocked} \wedge \circ\, \varphi) \parallel^{<} \psi &\leftrightarrow \exists\, \underline{u}.\ (\tau^{\underline{u}}_{\underline{x}''} \wedge \neg\, \mathbf{blocked} \wedge \circ\, ((\underline{x} = \underline{u} \wedge \varphi) \parallel \psi)) \\ (\tau \wedge \mathbf{blocked} \wedge \circ\, \varphi) \parallel^{<} \psi &\leftrightarrow \exists\, \underline{u}.\ (\tau^{\underline{x},\underline{u}}_{\underline{x}',\underline{x}''}) \wedge ((\underline{x} = \underline{u} \wedge \varphi) \parallel^{<}_{b} \psi) \end{aligned}$$

where $\underline{x} = \mathrm{free}(\tau)$. The first equivalence executes an unblocked transition τ according to Case 2 of Definition 1.4. The second equivalence considers the case where the first

$$
\begin{array}{ll}
\varphi \parallel_b^< (\psi_1 \vee \psi_2) & \leftrightarrow \varphi \parallel_b^< \psi_1 \vee \varphi \parallel_b^< \psi_2 \\
\varphi \parallel_b^< (\exists\, v.\ \psi) & \leftrightarrow \exists\, v_0.\ (\varphi \parallel_b^< \psi_v^{v_0}) \text{ where } v_0 \notin (\mathrm{free}(\psi) \setminus \{v\}) \cup \mathrm{free}(\varphi) \\
\varphi \parallel_b^< (\tau \wedge \mathbf{last}) & \leftrightarrow \mathit{false} \\
\varphi \parallel_b^< (\tau \wedge \neg\, \mathbf{blocked} \wedge \circ\, \psi) & \leftrightarrow \exists\, \underline{u}.\ \tau_{\underline{x''}}^{\underline{u}} \wedge \neg\, \mathbf{blocked} \wedge \circ\, (\varphi \parallel (\underline{x} = \underline{u} \wedge \psi)) \\
\varphi \parallel_b^< (\tau \wedge \mathbf{blocked} \wedge \circ\, \psi) & \leftrightarrow \exists\, \underline{u}.\ \tau_{\underline{x'},\underline{x''}}^{\underline{x},\underline{u}} \wedge \mathbf{blocked} \wedge \circ\, (\varphi \parallel (\underline{x} = \underline{u} \wedge \psi))
\end{array}
$$

Figure A.1.: Blocked Transition of Component 1.

component is blocked according to Case 3 of Definition 1.4. In this case, a blocked stutter transition of the first component is executed and the second component is also executed according to Figure A.1. Both equivalences above introduce new static variables $\underline{u}$ that store the values $I_1(1)(\underline{x})$ which can be referenced as $\underline{x''}$ in τ, see Definition 1.4. This is necessary since in the resulting global interval I, state $I(1)$ after the first global environment transition may be different from state $I_1(1)$ after the first local environment transition.

Index